Endbericht der Projektgruppe 423

DEVISOR

Lehrstuhl für
Graphische Systeme
Fachbereich Informatik

Lehrstuhl für Angewandte
Mathematik und Numerik
Fachbereich Mathematik

Universität Dortmund

WiSe 2002/03, SoSe 2003

Teilnehmer
Hendrik Becker
Christian Engels
Markus Glatter
Dominik Göddeke
Eduard Heinle
Mathias Kowalzik
Patrick Otto
Wissam Ousseili
Thomas Rohkämper
Mathias Schwenke
Nicole Skaradzinski
Tom Vollerthun
Betreuer
Claus-Peter Alberts
Jörg Ayasse
Christian Becker

Inhaltsverzeichnis

Kapitel 1

Einleitung

Motivation, Aufgabenstellung und Einordnung

1.1 Motivation und Projektidee

Um Naturvorgänge verstehen zu können, gibt es zwei grundlegende Ansätze: das Experiment und die Simulation. Während man beim Experiment versucht, die Wirklichkeit in verkleinertem Maßstab vereinfacht nachzubilden, z.B. in einem Windkanal, und damit Erkenntnisse zu gewinnen, geht die Simulation von einem abstrakten Ansatz aus. Man versucht, aus Naturvorgängen eine mathematische Beschreibung zu gewinnen, das sogenannte Modell. Dieses Modell kann z.B. aus einer Reihe von Differentialgleichungen bestehen, so beschreiben die sogenannten Navier-Stokes-Gleichungen Strömungsvorgänge (s. Kap. 2.2). Um nun aus dem Modell Vorhersagen für die Wirklichkeit ableiten zu können, ist es notwendig, das mathematische Modellproblem zu lösen. Während es möglich ist, für einfache Probleme die Lösung noch geschlossen, d.h. exakt, zu bestimmen, ist dies für komplexe realistische Probleme nicht mehr möglich. In diesem Fall werden Verfahren der numerischen Simulation eingesetzt. Dabei wird die Lösung nicht exakt ermittelt, sondern man nähert sich der tatsächlichen Lösung in mehreren Schritten immer genauer an. Ein wesentlicher Ansatz der numerischen Simulation sind Finite-Elemente-Methoden (FEM, zur Einführung s. [32], Kap. 1-4 und 2.2).
Ein typischer Arbeitszyklus beim Einsatz eines numerischen Simulationssystems sieht folgendermaßen aus:

1. Definition des Rechengebietes (Gitters), über dem die Berechnung ausgeführt wird (Abb. 1.1).
2. Editieren diverser Konfigurationsdateien, um die Simulationsparameter festzulegen.
3. Starten der Simulation.
4. Kontrolle der Simulation und Abschätzung, ob die Simulation erfolgreich verläuft.
5. Interpretation und Visualisierung der Ergebnisse (Abb. 1.2).

Die genannten Schritte werden so lange wiederholt, bis das Ergebnis zufriedenstellend ist oder der Anwender aufgibt.

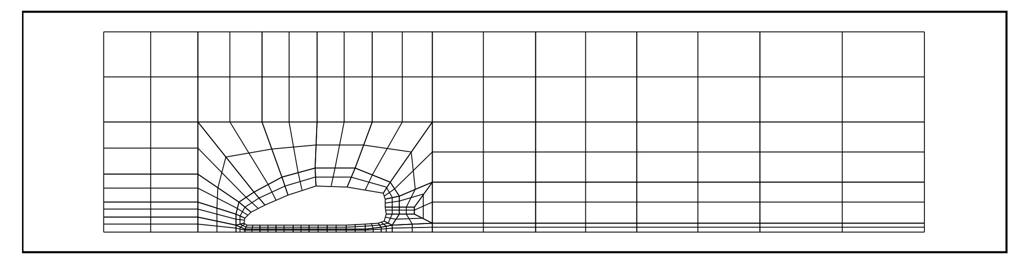


Abbildung 1.1: Beispielgitter für eine zweidimensionale Fahrzeugkonfiguration, die das Fahrzeug und einen Ausschnitt des umgebenden Raums umfaßt. Die Fahrzeugkontur ist unten links erkennbar.

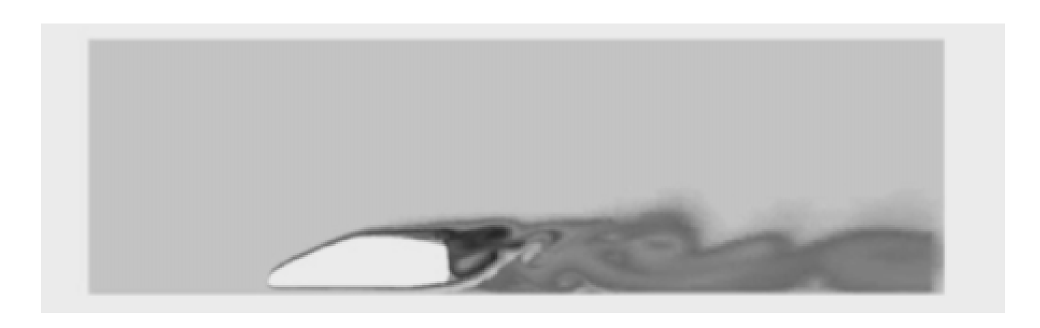


Abbildung 1.2: Visualisierung einer Fahrzeugumströmung, wie sie auf Grundlage der Konfiguration aus Abbildung 1.1 berechnet werden kann. Die Strömung ist per Shaded-Plot-Verfahren visualisiert. Erkennbar ist die Temperatur, je dunkler eine Stelle, desto wärmer ist sie. Am Anfang der Simulation ist nur die Fahrzeugkontur beheizt. Der Windstrom kommt gleichmäßig vom linken Rand. Am Verlauf der Temperatur ist das Strömungsverhalten erkennbar.

Diese Suche durch Versuch und Irrtum kann zahlreiche Simulationsläufe mit unterschiedlichen Parametern zur Folge haben, die dementsprechend zeit- und kostenaufwendig sind. Hinzu kommt, daß der Datenaustausch zwischen den leistungsfähigen Software-Werkzeugen, die heute für die einzelnen Schritte zur Verfügung stehen, wie Gittereditoren für Schritt 1, Simulationsprogramme für Schritt 2 und Visualisierungssoftware für Schritt 4, häufig nicht eng gekoppelt ist, da diese Werkzeuge unterschiedlichen Ursprung haben. Ein weiteres Problem besteht darin, daß aufgrund der benötigten Rechenleistung unterschiedliche Rechner zum Einsatz kommen, beispielsweise große Parallelrechner für die Simulation und leistungsfähige Graphik-Workstations für die Visualisierung, deren Kommunikation häufig nicht transparent ist.

1.2 Einordnung des Projekts

Es gibt durchaus viele Umgebungen, die numerische Strömungssimulationen gewissermaßen „unter einem Dach” anbieten. Diese Systeme haben jedoch alle einen prinzipiellen Nachteil: Nicht alle Komponenten entsprechen den Wünschen des Anwenders hinsichtlich Leistungsfähigkeit und Bedienbarkeit. Erschwerend kommt hinzu, daß der modulare Ansatz bei diesen

Paketen nicht berücksichtigt wurde, der einfache Austausch einer Komponente gegen eine andere – als naheliegende Lösung des Problems – ist nur durch den Hersteller möglich. Nachteile existierender Ansätze lassen sich wie folgt zusammenfassen:

- Oft sind die mathematischen Komponenten nicht auf dem neuesten Stand der Forschung, wie dies bei vielen kommerziellen Codes (ein Beispiel hierfür ist das auf MATLAB basierende FEMLAB [9]) zu beobachten ist.
- Der Datenaustausch zwischen leistungsfähigeren Komponenten scheitert an proprietären, nicht offen gelegten Formaten.
- Leistungsfähige Pakete sind oft universitäre Entwicklungen und zeichnen sich häufig nicht durch Benutzerfreundlichkeit und eine zumutbare Einarbeitungszeit aus, da auf die Entwicklung von intuitiven Benutzerschnittstellen dort kein Wert gelegt wird.
- Relativ leistungsfähige kommerzielle Pakete wiederum sind für viele Anwender einfach unbezahlbar, ein Beispiel ist die Software FLUENT [12].
- Die Netzwerktauglichkeit ist auch nicht selbstverständlich, zusätzlich zu der langen Laufzeit durch veraltete Verfahren kommt bei einer Simulation noch hinzu, daß sie auf dem heimischen PC durchgeführt werden muß.

Das von der Projektgruppe entwickelte System versucht, durch die Verfolgung eines streng modularen Ansatzes diese Probleme zu lösen: Im System sind alle Komponenten enthalten, die der Anwender benötigt, um direkt praktisch arbeiten zu können, so ein Gittereditor, ein Kontrollmodul und eine Visualisierungskomponente. Für die Durchführung der eigentlichen Simulation wurden beispielhaft die umfangreichen Numerikpakete FEATFLOW und FEAST (vgl. [40, 3, 37]) angebunden. Beide sind ohne Einschränkung im Quelltext frei verfügbar.
Mit ein wenig Programmiererfahrung ist es jedem Anwender möglich, gerade weil das von der Projektgruppe entwickelte System ebenfalls *open source* ist und unter der General Public Licence (GPL, s. E und [14]) veröffentlicht ist, beliebige Module zu ergänzen oder auszutauschen.
Weiter bieten kommerzielle Lösungen keinen Ausweg aus dem oben zitierten Versuch-und-Irrtum-Problem. Selbst kleine Änderungen der Simulationsparameter resultieren in einem neuen Berechnungszyklus, ältere Ergebnisse können nicht weiterverwendet werden. Es gibt zwar schon lange Ansätze zum *computational steering* (s. Kap. 2.1.3), d.h. zur interaktiven Kontrolle eines in der Ausführung befindlichen Rechenprozesses, aber diese Ideen sind bisher nicht auf den Bereich der numerischen Strömungssimulation übertragen worden. Die Darstellung von Zwischenergebnissen stellt dabei kein Problem dar, interessant ist die Neukonfiguration des Berechnungsprozesses, ohne wieder von vorne beginnen zu müssen.

1.3 Aufgabenstellung

Der in der Projektgruppe verfolgte Weg versucht, die genannten Schwierigkeiten zu mindern. Das DEVISOR – *Design and Visualization Software Ressource* getaufte System verfolgt dazu einen streng modularen Ansatz: Von einem zentralen Kontrollmodul aus kann der Benutzer alle Simulationsparameter über eine komfortable graphische Schnittstelle einstellen und den

Simulationsverlauf verfolgen. Der DEVISOR bietet mit der Integration der leistungsfähigen Pakete FEATFLOW und FEAST und den bereitgestellten Modulen zur Gittereditierung und Visualisierung bereits genug Werkzeuge an, um ohne lange Einarbeitungszeit direkt praktisch eigene Simulationen durchführen zu können. Der modulare Ansatz erlaubt weiterhin, flexibel ohne das existierende System ändern zu müssen, weitere Pakete anzubinden. Vorstellbar sind hier zusätzliche Numerikpakete (beispielsweise speziell für granulare Flußvorgänge) oder Werkzeuge zur automatischen Gittergenerierung, die im existierenden Editor noch nachbereitet werden können.
Eine weitere Kernkomponente des Systems ist die Verteiltheit: In praktischen Situationen möchte der Anwender gerne bequem am Schreibtisch die Konfiguration eines Simulationslaufs vornehmen und den Verlauf verfolgen. Der Schreibtisch-PC ist aber bei weitem nicht leistungsfähig genug, die eigentliche Berechnung sollte also besser auf einem Supercomputer und die Visualisierung der Ergebnisse auf einer Grafik-Workstation durchgeführt werden. Dieses Szenario ist mit dem DEVISOR leicht zu realisieren.
Die Realisierbarkeit eines *computational steering*-Ansatzes in Verbindung mit einer verteilten Anwendung wurde von der Projektgruppe ebenfalls prototypisch, sofern es ohne aufwendige Anpassungen des verwendeten Numerikpakets möglich war, untersucht und im Rahmen der DEVISOR-Spezifikation vorgenommen. Eine gute Umschreibung für das Ergebnis ist die *Kassettenrekorder-Metapher*: Der Benutzer kann eine Simulation nicht nur starten und beenden, sondern auch Funktionen wie Pause, Zurückspulen etc. anwenden, und in einer pausierten Berechnung jederzeit Zwischenergebnisse abfragen und visualisieren lassen. Auch die Neukonfiguration einzelner Parameter ist möglich.
Die konkreten Minimalziele der Projektgruppe wurden folgendermaßen festgelegt:

- dokumentierter Systementwurf,
- dokumentierte Implementierung des Systems und
- Demonstration der Funktionalität anhand einer vollständig durchgeführten Simulation.

Während der Planungs- und Spezifikationsphase (s. Kap. 4) wurden diese allgemeinen Ziele konkretisiert:

- Das Programmpaket DEVISOR besteht aus den Komponenten VISION zur Ergebnisdarstellung, GRID zur Gittereditierung, CONTROL zur Steuerung der einzelnen Module, NEXUS als Hintergrundserver und Schnittstelle zu den Simulationsprogrammen und NET zur Kommunikation der Module untereinander.
- Das Modul CONTROL bietet auf der Basis einer dynamisch generierten graphischen Bedienungsoberfläche die Möglichkeit, mit beliebigen Modulen Kontakt aufzunehmen, sie zu konfigurieren und ihre Berechnungen zu verfolgen (beispielsweise mit Hilfe eines Statistikplotters).
- GRID soll sowohl im Zwei- als auch im Dreidimensionalen die Generierung und das Editieren von Geometrie- und Gitterbeschreibungen ebenfalls in einer graphischen Oberfläche bieten. Rand- bzw. Geometriebeschreibungen sollen in Form von Polygonsegmenten, Oberflächentriangulierungen und einfachen Grundkörpern möglich sein bzw. aus CAD-Programmen importierbar sein. Rechengitter können durch Dreiecke und Tetraeder sowie Vierecke und Hexaeder definiert werden.

- VISION bietet die folgenden Visualisierungstechniken (s. Kap. 2.4) für zwei- und dreidimensionale Datensätze:
  - Isolines
  - Shaded plots
  - Glyphs
  - Particle traces
  - Cutlines, Cutplanes

  Zusätzlich stellt das Modul ein Interface zur Verfügung, das eine dreidimensionale Geometrie- und Gitterbeschreibung in einem komfortabel zu navigierenden 3D-Fenster bietet, insbesondere mit frei wählbaren Betrachterpositionen.

- NET stellt die Kommunikation der Module untereinander sicher und definiert dazu ein Übertragungsprotokoll.

- Da die Einarbeitung in die Numeriksoftware nicht explizit von den Teilnehmern gefordert war, implementiert einer der Betreuer den Hintergrundserver NEXUS und seine Anbindung an FEATFLOW und FEAST, die als beispielhafte Numerikpakete verwendet werden.

- Die oben angesprochene *Kassettenrekorder*-Funktionalität wird unterstützt. Um eine Neuimplementierung von FEATFLOW zu vermeiden, wird diese jedoch auf die Anwendung des sogenannten *Membranproblems* (s. Kap. 2.5) beschränkt.

- Die Funktionalität soll beispielhaft am *Membranproblem* demonstriert werden.

- Der Endbericht soll das komplette System dokumentieren und Bedienungsanleitungen sowie Tutorials zur Erweiterbarkeit enthalten.

1.4 Struktur des Berichts

Nach der allgemeinen Einleitung und der Erklärung benötigter Formalia, wie die genaue Aufgabenstellung und die von der Projektgruppe festgelegten Minimalziele, stellt das zweite Kapitel (s. Kap. 2) zunächst die benötigten theoretischen Grundlagen bereit:

- In Kapitel 2.1 werden wichtige Begriffe im Zusammenhang mit Computernetzwerken, zwei- und dreidimensionaler Grafikprogrammierung und *computational steering* bereitgestellt.

- Es folgt eine relativ umfangreiche Einführung in die mathematischen Grundlagen von numerischer Strömungssimulation (engl. *CFD, computational fluid dynamics*, s. Kap. 2.2): Für einige einfache Beispiele werden die mathematischen Modellierungen motiviert und präzise formuliert, insbesondere für das im Verlauf dieses Berichts immer wieder auftauchende *Membranproblem*. Auf Aspekte der Diskretisierung und in der Praxis verwendeter Löser wird eingegangen. Der Abschnitt schließt mit einer theoretischen Untersuchung der Anwendbarkeit von *computational steering*-Konzepten.

- Kapitel 2.3 führt in das wissenschaftliche Rechnen ein. Die praktische Umsetzung der mathematischen Verfahren in Simulationszyklen ist der Kernpunkt dieses Abschnitts.

- Kapitel 2.4 führt in die Grundlagen der wissenschaftlichen Visualisierung von skalaren und vektoriellen Daten ein. Da es eine Unmenge an Verfahren gibt, wird zunächst eine Katalogisierung vorgenommen, um dann einige typische Beispiele zu erläutern.

- Kapitel 2.5 beschreibt im Anschluß die verwendeten Datenformate, die die Projektgruppe teils zur Dateneingabe, teils zur Ausgabe der Programmergebnisse verwendet.

- Kapitel 2.6 schließt mit der Erläuterung der von der Projektgruppe verwendete Programme, sowohl Programmierumgebungen im weitesten Sinne als auch ein exemplarisches Paket zur numerischen Strömungssimulation. Es kann jedoch nur ein kurzer Überblick gegeben werden, für weiterführende Details wird auf Sekundärliteratur verwiesen.

Das darauf folgende Kapitel 3 stellt das von der Projektgruppe entwickelte System vor, und zwar zunächst das allgemeine Konzept und danach die konkrete Arbeitsweise der verschiedenen Teilprogramme: In diesem Abschnitt findet sich beispielsweise die Beschreibung wichtiger Algorithmen (sofern nicht bereits im Grundlagenkapitel 2 geschehen) und Klassen bzw. Methoden. Auch die Paketeinteilung wird hier begründet. Das Kapitel schließt mit der Darstellung der Integration der verschiedenen Module und liefert insbesondere detaillierte Kontroll- und Datenflußdiagramme.
Das Kapitel 4 befaßt sich mit der Durchführung des Projektes, insbesondere der Organisation innerhalb der Gruppe.
Der Anhang unterteilt sich in vier große Blöcke: Im Benutzerhandbuch (s. Kap. A) wird die Bedienung des Systems erklärt, beginnend mit der Installation und einigen für das Verständnis nötigen konzeptionellen Grundlagen. Es folgen eine kurze (das Schnelleinstieg-Tutorial) und eine ausgiebige Anleitung. Im Anschluß werden mögliche Erweiterungen diskutiert und in Form von Tutorials auch konkret vorgestellt: Einbau weiterer Visualisierungstechniken in das Modul VISION, Integration völlig neuer Module und nötige Anpassungen für die Verwendung einer anderen Numeriksoftware. Die Programmierschnittstelle (API) des Systems wird aus Platzgründen in diesem Bericht nur in Kurzform auf Klassenebene, nicht auf Methodenebene dokumentiert. Es schließt sich die Spezifikation der PG-eigenen Formate und Protokolle an, und der Bericht endet mit dem Abdruck der verwendeten Lizenz und einem umfassenden Literaturverzeichnis.

Kapitel 2

Grundlagen

Grundbegriffe und benötigtes Hintergrundwissen

2.1 Allgemeine Grundbegriffe

2.1.1 Computernetzwerke

Netzwerke

Ein Rechnernetz besteht aus mehreren verbundenen Computern. Die Art der Verbindung (ob per Kabel oder per Funk) ist unerheblich, wichtig ist, daß die Computer unabhängig voneinander existieren und das Netzwerk den gegenseitigen Nachrichten- und Informationsaustausch ermöglicht. Der Begriff Netzwerk beschreibt lediglich die physikalische Verbindung.

Verteilte Systeme

Verteilte Systeme setzen auf Netzwerken auf und bieten Funktionalitäten wie Kommunikation zwischen einzelnen oder allen Rechnern des Netzwerks, Ressourcenteilung, Lastverteilung und Sicherheitserhöhung. Ein Programm kann Teile seiner Arbeit auf verschiedene Rechner verteilen und die einzelnen Teile durch Nachrichtenaustausch synchronisieren.

Client-Server-Systeme

Client-Server-Systeme sind spezielle, verteilte Systeme, bei denen ein Dienstleister, der Server, mehrere Dienstnehmer (Clients) bedient. Die Clients können untereinander nur über den Umweg über den Server kommunizieren. Typischerweise startet der Server pro Verbindungsanfrage einen eigenen Prozeß, der exklusiv für die Kommunikation mit einem Client verantwortlich ist.

TCP/IP

TCP/IP ist ein Netzwerkprotokoll, das die Art der Nachrichtenübermittlung in einem Netzwerk beschreibt. Es definiert eindeutige Adressen für jeden Rechner im Netzwerk (sog. *IP-Adressen*) und teilt zu versendende Nachrichten in Pakete ein. Das Protokoll stellt sicher, daß Übertragungsfehler automatisch zu einer erneuten Versendung führen und gewährleistet die korrekte Reihenfolge ankommender Pakete unabhängig vom Weg, den sie durch das Netzwerk nehmen.

Sockets

Sockets sind die Ein- und Ausgabestellen eines Rechnerverbunds, und zwar auf Programmebene, nicht auf Hardwareebene. Jedes Programm kann u.U. viele Sockets an eine bestimmte Internetadresse *„binden"* und danach Nachrichten mit dieser Gegenstelle austauschen.

Prozesse und Threads

Ein Programm ist zunächst statisch. Sobald es gestartet wird, gehören neben dem auszuführenden Code noch Speicherbereiche, in dem Werte abgelegt werden können, ein Stack usw., zur *„Laufzeitversion"* des Programms. Einen solchen Verbund bezeichnet man als Prozeß. Verschiedene, gleichzeitig ablaufende Instanzen eines Programms sind verschiedene Prozesse. Soll innerhalb eines Prozesses zusätzliche Nebenläufigkeit erreicht werden (um z.B. nichtblockierend auf ein Lesen aus einer Socket zu warten), sind Threads, leichtgewichtige Prozesse, nötig. Computer mit mehreren Prozessoren ordnen typischerweise verschiedenen Prozessen verschiedene Prozessoren zu, bei Einprozessorsystemen (wie handelsüblichen PCs) werden verschiedene Prozesse bzw. Threads in Zeitscheiben eingeteilt und teilen sich, häufig konkurrierend, die CPU.
Für ausführlichere Informationen über Rechnernetzwerke und Betriebssysteme wird auf die Lehrbücher [34, 8, 36] verwiesen.

2.1.2 Computergrafik

Affine Transformationen

"Affine Transformation" ist ein Sammelbegriff für Translationen, Skalierungen und Rotationen.

Homogene Koordinaten Um alle affinen Transformationen einheitlich durch eine Matrix-Vektor-Multiplikation darstellen zu können, wird ein Punkt $\mathbf{q} = (x, y, z)^T \in \mathbb{R}^3$ überführt in den Punkt $\mathbf{p} = (x, y, z, w)^T = (x, y, z, 1)^T \in \mathbb{R}^4$. Dies entspricht einer Projektion auf die Ebene $\{w = 1\} \subset \mathbb{R}^4$. In homogenen Koordinaten können alle affinen Transformationen – im Gegensatz zu den kartesischen Koordinaten – einfach durch Matrix-Vektor-Multiplikationen dargestellt werden, die Hintereinanderausführung mehrerer Transformationen entspricht der Multiplikation ihrer entsprechenden Matrizen, und durch Multiplikation mit der inversen Matrix können sie rückgängig gemacht werden. Dies erlaubt insbesondere die Kaskadierung und damit die einfachere und effizientere Umsetzung in Hardware. Im folgenden werden alle Punkte in homogenen Koordinaten angenommen.

Translationen Eine Translation um den Vektor $\mathbf{t} = (t_1, t_2, t_3, 1)^T$ ist wie folgt definiert:

$$T(\mathbf{p}) := \begin{pmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{p}$$

Skalierungen Eine Skalierung um die Faktoren $\alpha, \beta, \gamma \in \mathbb{R}$ in x-,y- bzw. z-Richtung ist wie folgt definiert:

$$S(\mathbf{p}) := \begin{pmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{p}$$

Rotationen Eine Rotation des Vektors $\mathbf{p}$ um die x-Achse um den Winkel α lautet:

$$R_x(\mathbf{p}, \alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{p}$$

Eine Rotation des Vektors $\mathbf{p}$ um die y-Achse um den Winkel α lautet:

$$R_y(\mathbf{p}, \alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{p}$$

Eine Rotation des Vektors $\mathbf{p}$ um die z-Achse um den Winkel α lautet:

$$R_z(\mathbf{p}, \alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{p}$$

Eine beliebige Rotation im Raum kann durch eine Folge von Translationen und elementaren Rotationen um die Koordinatenachsen durchgeführt werden. Dies wird in [13] beschrieben.

Projektionen

Projektionen dienen dazu, dreidimensionale Objekte auf einer Ebene – beispielsweise auf dem Bildschirm – darzustellen. Hier wird das Prinzip beispielhaft für polygonale Flächen vorgestellt. O.B.d.A. seien die Objekte durch ihre Eckpunkte gegeben, so können alle Eckpunkte separat transformiert und wieder verbunden werden. Zur besseren Darstellung werden oft verdeckte Linien eliminiert oder außerhalb des gewünschten Bereichs liegende Teile an einem Sichtfenster bzw. Sichtvolumen abgeschnitten. Details dazu finden sich in [13].
Projektionen werden durch die folgenden Angaben spezifiziert:

- Betrachterstandpunkt $\mathbf{p}_v \in \mathbb{R}^3$,

- Projektionsebene $E = \mathbf{p} + \alpha\mathbf{a} + \beta\mathbf{b} \subset \mathbb{R}^2$,
- Projektionsrichtung(en), Projektoren $\mathbf{r}_i$,
- Sichtvolumen V_v durch die Angabe der vorderen und hinteren Clip-Ebene E_f und E_b.

Die Schnittpunkte der Projektoren durch die Eckpunkte des Objekts mit der Projektionsebene bilden die Projektion. Sie wird durch die Lösung eines linearen Gleichungssystems (eines Strahl-Ebene-Schnittproblems) durchgeführt.

Parallelprojektion Bei der Parallelprojektionen sind alle Projektoren modulo Parallelverschiebung gleich, die Strahlen durch alle Punkte sind parallel und schneiden sich nicht. Der Betrachterstandpunkt liegt formal im Unendlichen.

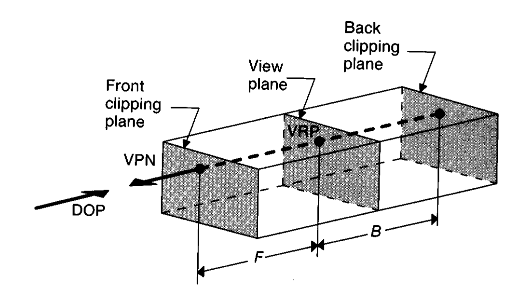


Abbildung 2.1: Parallelprojektion nach [13]

Perspektivische Projektion Bei der perspektivischen Projektion ist der Projektor eines Punktes der Strahl von einem festen endlichen Betrachterstandpunkt durch den jeweiligen Punkt.

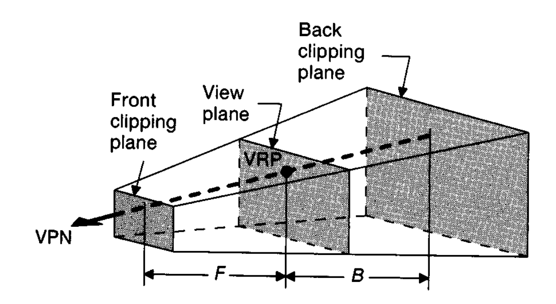


Abbildung 2.2: Perspektivische Projektion nach [13]

Beleuchtungsmodelle

Ein Beleuchtungsmodell beschreibt, wie eine dreidimensionale Szene zur Darstellung eingefärbt und beleuchtet werden soll. Dazu werden einzelnen Objekten gewisse Materialeigenschaften wie Reflexion oder Absorption einfallender Lichtstrahlen zugeordnet. Zusätzlich werden Lichtquellen definiert, diese können entweder punktförmig oder ausgedehnt sein. Auch

die Angabe, wie Lichtstrahlen, die sich in der Szene ausbreiten, mit wachsender Lauflänge an Intensität verlieren, ist Teil eines Beleuchtungsmodells.
In der Praxis hat sich im Wesentlichen das Beleuchtungsmodell von *Phong* durchgesetzt. Eine umfassende Erläuterung würde an dieser Stelle den Rahmen sprengen, daher wird auf die Literatur verwiesen, insbesondere auf [13].

Das Szenengraph-Konzept

Szenengraphen (engl. *scene graphs*) bieten eine Möglichkeit, 3D-Szenen kompakt zu beschreiben. Sie enthalten dazu nicht nur die reinen Geometrieobjekte, sondern zusätzlich Transformations-, Beleuchtungs- oder Materialattribute. Formal handelt es sich bei Szenengraphen um azyklische, gerichtete, zusammenhängende Graphen $G = (V, E)$ mit der Knotenmenge V und der Kantenmenge E. Alle Knoten sind attributiert:

- Ein Knoten ist speziell ausgezeichnet und heißt die Wurzel des Graphen. Von ihm gehen nur Kanten aus, er hat keine eingehenden Kanten. Hier kann beispielsweise der Betrachterstandpunkt angegeben werden.
- Gruppierungs- und Verzweigungsknoten dienen der Strukturierung von Teilgraphen.
- Geometrieknoten enthalten Beschreibungen von Szenenelementen, bspw. die Beschreibung eines Einheitsquaders oder, ein wenig komplexer, ein durch eine Triangulierung seiner Oberfläche gegebenes Objekt.
- Transformationsknoten beschreiben beliebige, geometrische Transformationen, so z.B. Skalierungen oder Translationen. Ein Transformationsknoten bezieht sich immer auf alle anderen Knoten, die von ihm aus durch einen gerichteten Weg aus Kanten erreicht werden können.
- Materialknoten werden Geometrieknoten zugeordnet und beschreiben bspw. Texturen, die auf die Objekte gelegt werden, oder Reflexionseigenschaften.
- Beleuchtungsknoten enthalten Informationen über Art, Ausdehnung und Richtung von Lichtquellen. Auch sie beziehen sich immer auf alle Nachfolgeknoten.

Ist ein Szenengraph einmal aufgebaut, erfordert seine Darstellung eine Breiten- oder Tiefensuche durch den Graphen. Treten auf einem Pfad zu einem Geometrieobjekt mehrere Transformationsknoten auf, so muß am zweiten Knoten die Transformation nur noch relativ zur ersten durchgeführt werden usw. Wenn zwei Objekte nicht durch einen gerichteten Pfad verbunden sind, so erfordert die Änderung eines Objekts keine Änderungen bei einem anderen.

2.1.3 Konzepte des computational steering

Computational steering kann man als interaktive Kontrolle eines in der Ausführung befindlichen Rechenprozesses verstehen. Während eines interaktiven Rechenprozesses werden Sequenzen von Spezifikationen, Berechnungen und Analysen durchgeführt. Für jede Anpassung des Berechnungsmodells muß dieser Prozeß wiederholt werden. *Computational steering* schließt diesen Kreis, indem einem Anwender ermöglicht wird, auf Resultate zu reagieren,

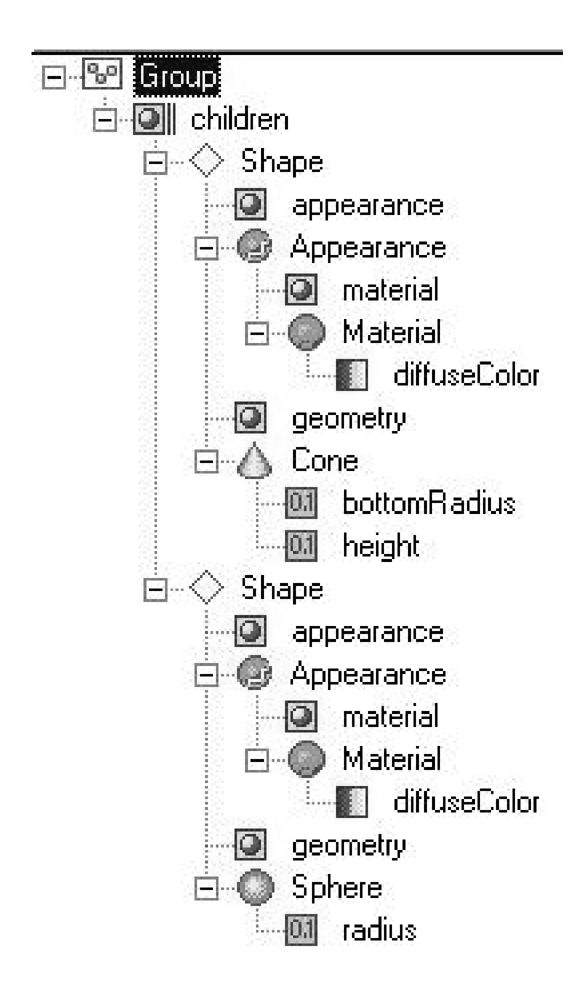


Abbildung 2.3: Beispiel für einen Szenengraphen

wenn sie aktuell vorliegen, indem er interaktiv die Eingabeparameter manipulieren kann. *Computational steering* erhöht die Produktivität, indem die Zeit, die normalerweise zwischen der Änderung von Parametern und der Visualisierung der Resultate auftritt, wesentlich reduziert wird. Es ermöglicht dem Anwender eine „Was-wäre-wenn”-Analyse. Da Änderungen von Parametern nun zeitnäher beobachtet werden können, kann der Forscher Zusammenhänge zwischen Ursache und Wirkung besser erkennen.

Computational steering ist in Bereichen der Computerwissenschaften in weitem Umfang einsetzbar. Weitgefaßt kann man alle interaktiven Berechnungsprozesse als eine Form von *computational steering* ansehen. Beispiele hierfür wären das zeilenweise Debuggen von Quellcode, die interaktive wissenschaftliche Visualisierung oder auch Computerspiele. In diesem Bereich können grundsätzlich drei Hauptanwendungen von *computational steering* festgestellt werden: *Modellerforschung, Experimentieren mit Algorithmen*, sowie die *Leistungsoptimierung*.

Bei der Modellerforschung wird *computational steering* dazu benutzt, den Parameterraum zu erforschen und dessen Auswirkung auf die Simulation zu beobachten, um neue Einsichten über die Simulation zu erhalten. *Computational steering* erlaubt beim Experimentieren mit Algorithmen die Anpassung eines Algorithmus während der Laufzeit, um zum Beispiel verschiedene

Methoden zur Berechnung numerischer Probleme auszuprobieren. Die Optimierung der Leistung einer Anwendung besteht zum Beispiel darin, *computational steering* so einzusetzen, daß man die Lastverteilung eines Parallelsystem interaktiv überprüfen und verändern kann.
Eine *computational steering*-Umgebung besteht aus drei Hauptkomponenten: dem Benutzerinterface, der Anwendung und der Komponente für Kommunikation und Datentransfer zwischen dem Benutzerinterface und der Anwendung. Diese Komponenten können in getrennten Prozessen ablaufen und damit auf verteilten Systemen implementiert werden, oder sie können vereint als einzelner Prozeß auf einer Maschine implementiert werden.
Die Anwendung selbst könnte in verteilten Prozessen realisiert sein. Außerdem ist der Prozeß des *computational steering* nicht auf eine Anwendung oder einen Benutzer beschränkt. Verschiedene Benutzer könnten eine oder mehrere Anwendungen in einem gemeinschaftlichen Steuerungsprozeß simultan manipulieren.
Die Umgebung muß die Anwendung überwachen und von dieser angeforderte Informationen entnehmen, um sie dem Benutzer zu präsentieren. Die Art der Informationen kann sich durch den Umfang der *computational steering*-Anwendung unterscheiden. Bei der Modellerforschung ist der Benutzer hauptsächlich an den Ein- und Ausgabeparametern interessiert.
Das Experimentieren mit Algorithmen würde es erfordern, dem Benutzer die Programmstruktur der Anwendung offenzulegen. Um eine Optimierung der Leistung durchführen zu können, muß der Benutzer über die Konfiguration und den Fortschritt des Programms informiert sein. Zum Beispiel muß bei parallelen oder verteilten Systemen die Verteilung der Anwendung auf die verschiedenen Prozessoren oder Plattformen bekannt sein, sowie die Netz- und Prozessorlast. Um Steuerungsaktionen starten zu können, muß die Umgebung Zugang zu den Eingabeparametern der Anwendung, dem ausführenden Code oder der Konfiguration der Anwendung haben. Der Zugang zu den überwachten Informationen und den Steuerungselementen kann asynchron oder synchron zur Anwendung ablaufen. Zum Beispiel könnten die Daten der Anwendung nicht immer gültig und einige Eingabeparameter während einer Berechnung nicht veränderbar sein.
Da *computational steering* ein in hohem Maße interaktiver Prozeß ist, bekommt das Benutzerinterface eine entscheidende Bedeutung in der *computational-steering*-Umgebung. Das Benutzerinterface hat zwei Aufgaben. Zuerst muß die aus der Anwendung gewonnene Information dem Benutzer präsentiert werden. Idealerweise geschieht dies durch die Visualisierung, wobei dies auch durch eine einfache Textanzeige geschehen kann. Die zweite Aufgabe des Benutzerinterfaces ist es, dem Benutzer die Manipulation der Steuerungselemente für die Anwendung zu ermöglichen. Diese Manipulationen können durch Textfelder, die Benutzung von einfachen, graphischen Interaktionsobjekten, wie Schiebereglern oder Knöpfen, mittels direkter Manipulation in der Visualisierung oder durch komplexe (3D) Eingabe-Interaktionsobjekte geschehen.
Beispiele für solche Umgebungen sind das Visualisierungstool AVS [2] und der IRIS EXPLORER [25]. Theoretische Untersuchungen finden sich im Grundlagenartikel [20].

2.2 Mathematischer Hintergrund

Dieses Kapitel beschreibt die mathematischen Grundlagen von numerischer Strömungssimulation:

1. Modellbildung: Hier wird ein physikalisches Phänomen durch eine Reihe von Differen-

tialgleichungen beschrieben. Um verschiedenste Probleme mit einheitlichen Verfahren lösen zu können, werden die Modelle mit Hilfe von Differentialgleichungen formuliert.

2. Diskretisierung: Das Modell wird nun diskretisiert, d.h. in eine für den Computer verarbeitbare Form gebracht. Dies liefert typischerweise große lineare Gleichungssysteme.

3. Berechnung: Diese Gleichungssysteme werden abschließend gelöst.

Es werden folgende Notationen verwendet:
Es werden Gebiete $\Omega \subseteq \mathbb{R}^d$ betrachtet, für $d = 1, 2, 3$ mit dem Rand $\partial\Omega$. Der senkrecht zum äußeren Rand stehende Normalenvektor von $\partial\Omega$ ist $\mathbf{n}$, der Tangentenvektor $\mathbf{t}$ (s. Abb. 2.4).

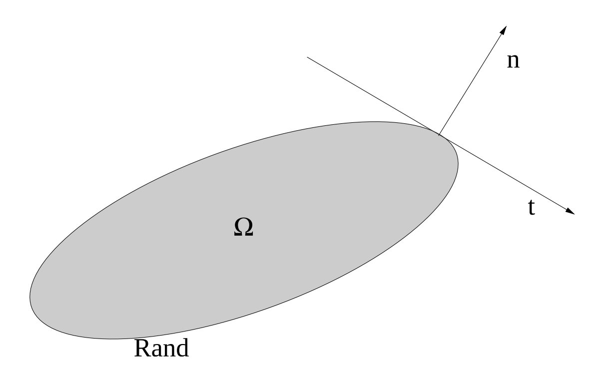


Abbildung 2.4: Definition eines Gebiets

Ortspunkte werden mit $\mathbf{x} = (x_1, x_2, x_3)^T$, Zeitpunkte mit t bezeichnet. Für d-dimensionale Vektoren $\mathbf{a}, \mathbf{b}$ wird das übliche euklidische Skalarprodukt mit $\mathbf{a} \cdot \mathbf{b}$, und die euklidische Norm mit $||\mathbf{a}||$ bezeichnet.

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &:= \sum_{i=1}^{d} a_i b_i \\ ||\mathbf{a}|| &:= \sqrt{\mathbf{a} \cdot \mathbf{a}} \end{aligned}$$

Funktionen treten entweder skalarwertig $u = u(\mathbf{x})$ oder $u = u(\mathbf{x}, t)$ oder vektorwertig $\mathbf{u} = (u_1, \ldots, u_d)(\mathbf{x}, t)$ auf, ausführlicher

$$\mathbf{u} = \begin{pmatrix} u_1(\mathbf{x}, t) \\ \vdots \\ u_d(\mathbf{x}, t) \end{pmatrix}$$

Partielle Ableitungen, d.h. Ableitungen nach einer der vorkommenden Variablen, werden wie folgt geschrieben:

$$\begin{aligned} \partial_t \mathbf{u} &:= \frac{\partial \mathbf{u}}{\partial t} \\ u_{x_i} := \partial_i \mathbf{u} &:= \frac{\partial \mathbf{u}}{\partial x_i}. \end{aligned}$$

Mit dem Nabla-Operator ∇ werden der Gradient (Vektor aller partiellen Ableitungen) einer skalaren sowie die Divergenz einer Vektorfunktion geschrieben als

$$\text{grad}\, u := \nabla u := (\partial_1 u, \ldots, \partial_d u)^T$$

$$\text{div}\, \mathbf{u} := \nabla \cdot \mathbf{u} := \partial_1 u_1 + \cdots + \partial_d u_d.$$

Zu einem Vektor $\mathbf{b} \in \mathbb{R}^d$ wird die Ableitung in Richtung $\mathbf{b}$ mit $\partial_b u := \mathbf{b} \cdot \nabla u$ bezeichnet. Entsprechend ist $\partial_n u := \mathbf{n} \cdot \nabla u$ die Ableitung in Richtung der äußeren Normalen auf dem Gebietsrand.

Die Kombination von Divergenz- und Gradientenoperator ergibt den *Laplace-Operator*

$$\Delta u = \nabla \cdot (\nabla u) = \partial_1^2 u + \cdots + \partial_d^2 u.$$

Für weitergehende Informationen sei auf diverse Lehrbücher, wie z.B. [33], oder entsprechende Vorlesungsskripte, wie z.B. [27], verwiesen.

2.2.1 Modellierung

Ziel der Modellierung ist es, für Naturvorgänge eine – oft stark vereinfachte – mathematische Beschreibung zu finden, d.h. Vorgänge wie Diffusion, Stofftransport oder Temperaturverteilung mit Hilfe mathematischer Gleichungen zu beschreiben. Dies ist ein höchst kreativer Akt, der viel Erfahrung erfordert und für den es kein Standardkochrezept gibt. Deshalb sollen an dieser Stelle nur drei exemplarische Vorgänge dargestellt werden. Um alle Modelle später gleich behandeln zu können, werden sie in Form von Differentialgleichungen beschrieben.

Beispiele

1. Wellengleichung Schwingende Ausbreitungsvorgänge sind z.B. die Ausbreitung einer Wasserwelle (lokale Störung der Wasseroberfläche) oder eines Geräuschs (lokale Störung der Luftdichte). Es erscheint sinnvoll (hier im Eindimensionalen), diese im einfachsten Fall durch eine Funktion $u = u(x,t)$ im Ort x und der Zeit t der Form $u(x,t) = \sin(x)\sin(t)$ zu beschreiben. Die Welle wird als Sinusfunktion im Ort modelliert, zusätzlich beeinflußt durch eine Sinusfunktion in der Zeit. Diese genügt der Differentialgleichung

$$\partial_t^2 u = \partial_x^2 u. \tag{2.1}$$

Beweis: für den skalaren Fall:

$$\partial_t^2 \sin(x)\sin(t) = \partial_t \sin(x)\cos(t) = -\sin(x)\sin(t)$$

$$\partial_x^2 \sin(x)\sin(t) = \partial_x \cos(x)\sin(t) = -\sin(x)\sin(t)$$

□

Gleichungen dieser Form heißen auch *hyperbolisch*.

2. Wärmeleitungsgleichung Andere Ausbreitungsvorgänge sind dadurch gekennzeichnet, daß lokale Störungen nicht schwingend, sondern "diffundierend" und sich abschwächend fortpflanzen, z.B. die Temperaturausbreitung in einem Leiter (Wärmeleitung) oder die Verteilung einer Dichtekonzentration in einer Flüssigkeit (Stofftransport). Zur Beschreibung solcher Vorgänge dienen Funktionen der Form $u(x,t) = \sin(x)e^{-t}$, hier als Beispiel im Eindimensionalen. Die Sinus-Funktion spiegelt die (im Ort ungedämpfte) Wellenbewegung wider, während die e-Funktion für eine Dämpfung in der Zeit sorgt. Diese Funktion genügt der Differentialgleichung

$$\partial_t u = \partial_x^2 u. \tag{2.2}$$

Diese Gleichungstypen werden als *parabolisch* bezeichnet.

3. Auslenkung einer Membran Im folgenden soll eine Membran betrachtet werden, die am Rand eingespannt ist, unter Belastung durch eine vertikale Kraft (Abbildung 2.5).

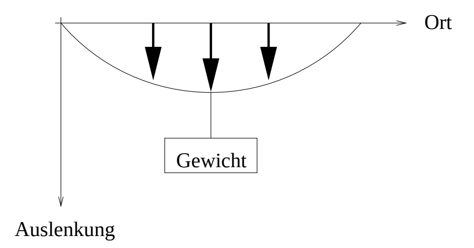


Abbildung 2.5: Auslenkung einer Membran

Sei

- die Kraftdichte $f(x,y)$ (gegeben),
- die Auslenkung $u(x,y)$ (gesucht) und
- die Einspannbedingung $u(x,y) = 0$ auf $\partial\Omega$, mit $x, y \in \mathbb{R}$

Man betrachtet die Energie J des Systems,

$$J(u) = J_1(u) + J_2(u).$$

Dabei sei J_1 die Spannungsenergie und J_2 die potentielle Energie. J_1 ist proportional zur Oberflächenänderung:

- Oberfläche vor Auslenkung: $\int_\Omega 1\,dx\,dy$,
- Oberfläche nach Auslenkung: $\int_\Omega \sqrt{1+u_x^2+u_y^2}\,dx\,dy$, mit u_x, u_y Ableitungen nach x bzw. y.

Zusammen erhält man nach der Hinzunahme eines Elastizitätfaktors α:

$$J_1(u) = \alpha \int_\Omega \sqrt{1+u_x^2+u_y^2} - 1\,dx\,dy.$$

Man nimmt kleine Auslenkungen an und verwendet deshalb die Taylorentwicklung (Entwicklung nach Potenzen von x und y zur näherungsweisen Darstellung der Funktion, je mehr Potenzen verwendet werden, desto genauer ist die Näherung):

$$\begin{aligned} \sqrt{1+u_x^2+u_y^2}-1 &\approx 1+\frac{1}{2}u_x^2+\frac{1}{2}u_y^2-1=\frac{1}{2}|\nabla u|^2 \\ \Rightarrow J_1(u) &\approx \alpha/2\int_\Omega |\nabla u|^2\,dx\,dy \end{aligned}$$

Für die potentielle Energie J_2 gilt:

$$J_2(u)=-\int_\Omega f u\,dx\,dy$$

Die Gesamtenergie ist nun

$$J(u)\approx \alpha/2\int_\Omega |\nabla u|^2\,dx\,dy-\int_\Omega f u\,dx\,dy.$$

Eine Funktion $u=u(x,y)$ ist die gesuchte Auslenkung, falls sie das Prinzip der minimalen Energie erfüllt:

$$J(u)\leq J(v) \qquad \text{für alle zulässigen Auslenkungen } v.$$

Also gilt auch:

$$\text{Für alle } \varepsilon>0: J(u)\leq J(u+\varepsilon v)\ \forall v$$

$$\Rightarrow \frac{d}{d\epsilon}J(u+\varepsilon v)\bigg|_{\varepsilon=0}=0\ \forall v.$$

Dies ist mit folgenden elementaren Umformungen äquivalent zu Gleichung (2.3).

$$\begin{aligned} & \tfrac{d}{d\varepsilon}J(u+\varepsilon v)\ \forall v \\ = & \tfrac{d}{d\varepsilon}\textstyle\int_\Omega |\nabla(u+\varepsilon v)|^2\,dx\,dy-\int_\Omega f\,(u+\varepsilon v)\,dx\,dy\ \forall v \\ = & \tfrac{d}{d\varepsilon}\textstyle\int_\Omega \partial_x(u+\varepsilon v)\partial_x(u+\varepsilon v)+\partial_y(u+\varepsilon v)\partial_y(u+\varepsilon v)-f\,(u+\varepsilon v)\,dx\,dy\ \forall v \\ = & \tfrac{d}{d\varepsilon}\textstyle\int_\Omega \partial_x^2 u+2\partial_x u\partial_x\varepsilon v+\partial_x^2\varepsilon v+\partial_y^2 u \\ & +2\partial_y u\partial_y\varepsilon v+\partial_y^2\varepsilon v-f\,(u+\varepsilon v)\,dx\,dy\ \forall v \\ = & \textstyle\int_\Omega 2\partial_x u\partial_x v+2\varepsilon\partial_x^2+2\partial_y u\partial_y v+2\varepsilon\partial_y^2-f\,v\,dx\,dy\ \forall v \end{aligned}$$

Setzen von $\varepsilon=0$ und Anwenden der ∇-Definition führen zu folgender Gleichung:

$$\alpha\int_\Omega \nabla u\nabla v-\int_\Omega f\,v=0\ \forall v. \tag{2.3}$$

Mit der Greenschen Formel (s. [16]) ergibt sich:

$$-\alpha \int_\Omega \Delta uv \, dx \, dy + \alpha \int_{\partial\Omega} \partial_n uv \, ds - \int_\Omega fv \, dx \, dy = 0 \; \forall v.$$

Das Randintegral ist Null, da auf dem Rand Nullwerte vorausgesetzt sind.
Es bleibt

$$\int_\Omega (-\alpha\Delta u - f)v \, dx \, dy = 0.$$

Da dies für alle Funktionen v gilt, muß die Klammer nach dem DuBois-Reynold-Lemma (s. [16]) bereits die Nullfunktion sein. Man erhält die Gleichung

$$-\alpha\Delta u = f \quad \text{in } \Omega \text{ mit Nullrandbedingungen.} \tag{2.4}$$

Gleichungen dieses Typs heißen speziell *Poissonprobleme*, allgemein gehören sie zum dritten Typ der Klassifikation, *elliptisch*.

4. Strömungsprobleme Um die Grundgleichungen der Strömungsprobleme zu beschreiben, sind zunächst zwei Koordinatensysteme zu unterscheiden:

eulersche Koordinaten beziehen sich auf ein festes Bezugssystem, d.h. der Beobachter sieht Teilchen durch ein festes Fenster an sich vorbeiziehen,

lagrangesche Koordinaten beziehen sich auf ein bewegtes Bezugssystem, d.h. der Beobachter sitzt direkt auf einer sich verbiegenden Metallplatte.

Man muß zwischen zwei verschiedenen Zeitableitungsbegriffen unterscheiden: Die materielle Zeitableitung du/dt bezeichnet die Änderungsrate eines sich bewegenden Fluidpartikels, während die lokale Zeitableitung $\partial u/\partial t$ die Änderungsrate in einem festes Punkt angibt. Mit Hilfe der Kettenregel ergibt sich der Zusammenhang

$$du/dt = \partial u/\partial t + \mathbf{v} \cdot \nabla u,$$

was aufintegriert über ein Kontrollvolumen V das Reynoldsche Transporttheorem (s. Formel 2.5) liefert.
Die Grundgleichungen der (klassischen) Kontinuumsmechanik basieren auf dem physikalischen Grundprinzip der „Erhaltung", d.h.: Zustandsgrößen, wie z.B. Massedichte $\rho(\mathbf{x}, t)$, totale Energie bzw. Temperatur $T(\mathbf{x}, t)$, Impuls $\rho(\mathbf{x}, t)\mathbf{v}(\mathbf{x}, t)$ u.s.w., werden als Dichtefunktionen beschrieben, deren Integrale über beliebige bewegte Kontrollvolumen sich beim Fehlen von äußeren Einflüssen nicht verändern.
Für die zeitliche Veränderung der Masse $m_V(t)$ eines solchen mit dem Geschwindigkeitsfeld $\mathbf{v}$ bewegten Volumens $V(t)$ mit der Oberfläche $S(t)$ gilt (sog. *Reynoldsches Transporttheorem*)

$$0 = \frac{dm_V}{dt} = \frac{d}{dt}\int_{V(t)} \rho(\mathbf{x}, t) \, dx = \int_{V(t)} \frac{\partial\rho(\mathbf{x}, t)}{\partial t} \, dV + \int_{S(t)} \rho\mathbf{v} \cdot \mathbf{n} \, dS. \tag{2.5}$$

Da dies für beliebige Volumen $V(t)$ gelten soll, ergibt sich für stetige Dichtefunktionen die folgende Erhaltungsgleichung 1. Ordnung (sog. „Kontinuitätsgleichung"):

$$\partial_t\rho + \nabla \cdot (\rho\mathbf{v}) = 0.$$

Auf analogem Wege erhält man aus dem Erhaltungssatz für die Temperatur unter Berücksichtigung von Quelltermen q und einem Wärmediffusionsfaktor $\kappa \in \mathbb{R}$ in einem ruhenden Medium ($\mathbf{v} \equiv 0$) die folgende Erhaltungsgleichung 2. Ordnung (sog. „Wärmeleitungsgleichung”):

$$\frac{\partial T}{\partial t} + (\mathbf{v} \cdot \nabla)T = \kappa \nabla^2 T + q$$

Physikalische Anschauung erfordert, daß Lösungen zu diesen Gleichungen bei physikalisch sinnvollen Anfangs- und Randbedingungen stets positiv sind: $\rho > 0,\ T > 0$. Die entsprechenden Erhaltungssätze für Dichte, Impuls und Energie führen unter geeigneten zusätzlichen Annahmen mit der Kontinuitätsgleichung auf die sogenannten *Navier-Stokes-Gleichungen* für inkompressible Flüssigkeiten, d.h. Flüssigkeiten, die auch unter Druck ihre Dichte und Temperatur nicht ändern:

$$\partial_t \mathbf{v} - \nu \Delta \mathbf{v} + \mathbf{v} \cdot \nabla \mathbf{v} + \nabla p = \mathbf{f}, \quad \nabla \cdot \mathbf{v} = 0. \tag{2.6}$$

2.2.2 Diskretisierung

Nachdem ein mathematisches Modell in Form von Differentialgleichungen vorliegt, ist die Frage, was man tun kann, um aus diesem eine Lösung zu gewinnen, um eine Vorhersage für den zugrundeliegenden Naturvorgang treffen zu können. Bei einfachen Gleichungen, wie zum Beispiel für die Wärmeleitungsgleichung, ist es vielleicht mit viel Erfahrung, Geduld und Intuition möglich, direkt durch scharfes „Daraufschauen” eine analytische Lösung zu bestimmen. Bei komplexeren Systemen, wie den Navier-Stokes-Gleichungen, ist dies nicht mehr möglich, daher ist man in diesem Fall auf ein numerisches Verfahren angewiesen.
Es gibt eine Reihe von Verfahren, wie z.B. das Differenzenverfahren, bei dem die Differentialoperatoren durch Differenzenquotienten ersetzt werden, wie z.B.

$$\partial_x f(x) \approx \frac{f(x+h) - f(x)}{h} \text{ für kleines } h.$$

Das Verfahren, um das es im folgenden etwas konkreter gehen wird, heißt Finite-Elemente-Verfahren.
Anders als beim Differenzenverfahren, bei dem die punktweise Gültigkeit der Gleichung gefordert wird, wird eine unter gewissen Zusatzforderungen äquivalente Variationsgleichung zugrunde gelegt, die eine Übereinstimmung nur in einem gewissen Mittel fordert.
Die Methode der Finiten Elemente soll an folgendem Beispiel betrachtet werden:

$$-u''(x) + b(x)u'(x) + c(x)u(x) = f(x) \text{ in } \Omega := (0,1), u(0) = u(1) = 0.$$

Dabei seien b, c und f gegebene, stetige Funktionen, und das Problem besitze eine Lösung, d.h. es existiere eine zweimal stetig differenzierbare Funktion $u \in C^2(\Omega) \cap C(\bar{\Omega})$, welche der Gleichung genügt. Damit genügt diese Lösung auch der folgenden Gleichung

$$\int_\Omega (-u'' + bu' + cu)v\, dx = \int_\Omega fv\, dx$$

mit beliebigen stetigen Funktionen v. Genügt umgekehrt eine Funktion $u \in C^2$ für beliebiges stetiges v dieser Beziehung, so genügt u auch der Differentialgleichung.

Weitergehende Untersuchungen zur Theorie bezüglich Existenz und Eindeutigkeit sollen an dieser Stelle ausbleiben, ausführliche Betrachtungen finden sich in [38].
Als konkretes Beispiel soll die schon bekannte Laplace-Gleichung

$$-\Delta u = f \text{ mit } u|_{\partial\Omega} = 0$$

dienen.
Ihre sogenannte schwache bzw. variationelle Formulierung lautet dann

$$\int_\Omega -\Delta uv \, d\Omega = \int_\Omega fv \, d\Omega \tag{2.7}$$

Durch partielle Integration ([16], Seite 85) erhält man die sogenannte Greensche Formel

$$\int \Delta uv = \int_{\partial\Omega} \partial_n uv \, dS - \int_\Omega \nabla u \nabla v \, d\Omega.$$

Der Δ-Operator wird quasi auf beide Funktionen "verteilt", wobei zusätzlich das Randintegral entsteht. Geht man von Nullrandbedingungen aus, so verschwindet dieses aber und es bleibt:

$$\int_\Omega \Delta uvd\Omega = -\int_\Omega \nabla u \nabla vd\Omega.$$

Daraus ergibt sich folgende Formulierung

$$\int_\Omega \nabla u \nabla v \, d\Omega = \int_\Omega fv \, d\Omega,$$

welche die Grundlage für die folgenden Ausführungen sein wird.
Um das Problem lösen zu können, betrachtet man eine näherungsweise Lösung, indem anstelle des zugrundeliegenden, im Fall der Anwendung auf Differentialgleichungen stets unendlichdimensionalen Raumes V, ein Teilraum $V_h \subset V$ mit $N := \dim V_h < +\infty$ gewählt wird, in dem die Lösung gesucht wird.
Die schwache Formulierung lautet dann

$$\int_\Omega \nabla u_h \nabla v_h \, d\Omega = \int_\Omega f_h v_h \, d\Omega. < F19 >$$

Da die Dimension von V_h endlich ist, gibt es endlich viele linear unabhängige Funktionen $\phi_i \in V_h, i = 1, \ldots, N$, die den Teilraum V_h aufspannen, d.h.

$$V_h = \{v_h : v_h(x) = \sum_{i=1}^N s_i \phi_i(x), \quad s_i \in \mathbb{R}\}.$$

Mit der Linearität der verwendeten Operatoren und f ist das Ausgangsproblem äquivalent zu folgendem:

$$\int_\Omega \nabla u_h \nabla \phi_i \, d\Omega = \int_\Omega f_h \phi_i \, d\Omega, \quad i = 1, \ldots, N.$$

Da die Lösung u_h in V_h liegen soll, besitzt sie ebenfalls eine Darstellung der Form

$$u_h(x) = \sum_{i=1}^{N} s_i \phi_i(x).$$

Einsetzten und Herausziehen der Summe ergibt:

$$\sum_{j=1}^{N} s_j \int_\Omega \nabla\phi_j \nabla\phi_i \, d\Omega = \int_\Omega f_h \phi_i \, d\Omega, \quad i = 1, \ldots, N.$$

Die Funktionen ϕ_i, ϕ_j heißen Ansatz- bzw. Testfunktionen. Obwohl es möglich ist, verschiedene Funktionenräume zu verwenden (sog. *Petrov-Galerkin-Verfahren*), wird meistens jedoch ein gemeinsamer Funktionenraum verwendet (sog. *Galerkin-Verfahren*).
Setzt man als Testfunktionen v die N Basisfunktionen ϕ_j ein, entsteht ein lineares Gleichungssystem der Form $A\mathbf{s} = \mathbf{b}$ mit den Unbekannten $\mathbf{s} = (s_1, ..., s_N)$, den Matrixeinträgen $a_{ij} = \int_\Omega \nabla\phi_j \nabla\phi_i \, d\Omega$ und der rechten Seite $\mathbf{b} = \int_\Omega f_h \phi_i \, d\Omega$ für $i = 1, \ldots, N$.
Folgende Fragen bleiben zu klären:

- Wahl der Funktionen ϕ_i und Zerlegung des Gebiets Ω,
- Berechnung der Integrale,
- Lösen des Gleichungssystems.

2.2.3 Wahl der Basisfunktionen und Zerlegung des Gebiets

Die Wahl der Basisfunktionen – auch Ansatz- oder Testfunktionen genannt – ist für die Realisierbarkeit und Effizienz des Verfahrens von großer Bedeutung. Im Unterschied zu klassischen Ansätzen mittels global einheitlich definierter Funktionen werden bei der Methode der Finiten Elemente stückweise definierte Funktionen – in der Regel Polynome – zugrunde gelegt. Die dabei erzeugten diskreten Probleme besitzen eine spezielle Struktur, und es lassen sich angepaßte Verfahren zu ihrer numerischen Behandlung angeben.
Die Methode der Finiten Elemente besitzt die folgenden drei typischen Merkmale:

- Zerlegung des Grundgebietes in geometrisch einfache Teilgebiete, z.B. Dreiecke und Rechtecke bei Problemen in der Ebene oder Tetraeder und Hexader bei Problemen im dreidimensionalen Raum,
- Definition von Ansatz- und Testfunktionen über den Teilgebieten und
- Einhaltung von Übergangsbedingungen bei den Ansatzfunktionen zur Sicherung gewünschter globaler Eigenschaften, wie z.B. der Stetigkeit.

Zerlegung des Gebiets Ω Die unterschiedlichen Typen von Gittern lassen sich in folgende Klassen einteilen:

1. Uniform strukturierte (kartesische) Gitter (engl. *cartesian grids*) bestehen aus Zellen, die alle gleich aufgebaut sind, beispielsweise Dreiecke, Tetraeder, Vierecke oder Hexaeder.

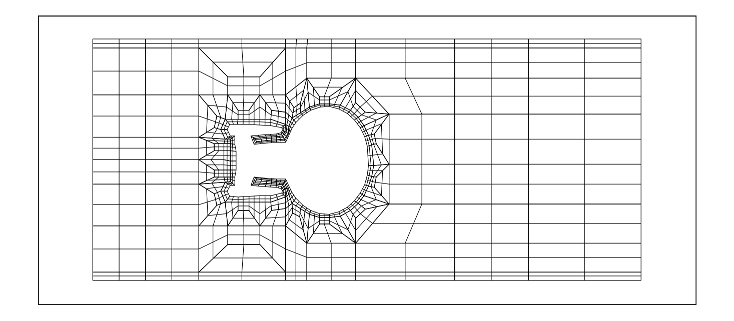


Abbildung 2.6: Beispielgitter aus Quaderelementen

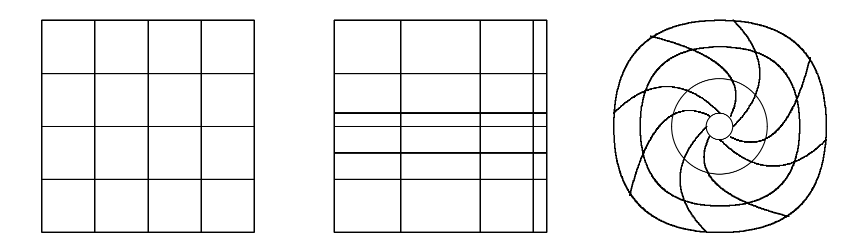


Abbildung 2.7: Beispiele für Gittertypen nach [24]: kartesisch, rechteckig bzw. strukturiert, allgemein

2. Rechteckig strukturierte Gitter (engl. *rectilinear grids*) bestehen aus Rechteckzellen, die aber in Breite und Höhe variieren können.
3. Strukturierte Gitter (engl. *stuctured, curvilinear grids*) sind Tetraeder- oder Hexaedergitter.
4. Allgemein strukturierte Gitter (engl. *general structured grids*) sind Gitter, die auf irgendeine Art und Weise strukturiert sind, beispielsweise durch Polygone oder Kreise etc.
5. Unstrukturierte Gitter (engl. *unstructured grids*) verwenden verschiedene Zelltypen in einem Gitter.

Definition eines Finiten Elements Die konkreten Finiten Elemente werden charakterisiert durch die Geometrie des Teilgebietes K (s.o.), die Anzahl, die Lage und die Art der Vorgaben für die Ansatzfunktionen.
Die unabhängig vorgebbaren Informationen werden in Anlehnung an die Mechanik „Freiheitsgrade" genannt. Der Einfluß der einzelnen Freiheitsgrade wird durch entsprechende Formfunktionen erfaßt.
Als Beispiel für den „Zoo" von Finiten Elementen soll an dieser Stelle ein einfaches lineares Viereckselement erläutert werden (s. Abbildung 2.8).
Die Freiheitsgrade für die Funktionswerte liegen in den Ecken. In der Praxis sind bilineare Ansatzfunktionen gebräuchlich, wichtig ist, daß sie in einer Ecke den Wert 1 annehmen und

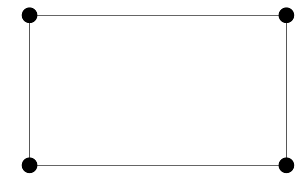


Abbildung 2.8: rechteckiges Finites Element

in den anderen 0. Für ein Einheitsrechteck, d.h. $\xi, \eta \in [0,1]$, können die Ansatzfunktionen folgendermaßen aussehen:

$$\begin{aligned}
\lambda_1 &= (1-\xi)(1-\eta), \\
\lambda_2 &= \xi(1-\eta), \\
\lambda_3 &= \xi\eta, \\
\lambda_4 &= (1-\xi)\eta.
\end{aligned}$$

Die i-te Funktion ist dabei im i-ten Eckpunkt gleich Eins, in den anderen Eckpunkten gleich Null und dazwischen hat sie einen linearen Verlauf. Abbildung 2.9 zeigt exemplarisch zwei dieser Funktionen.

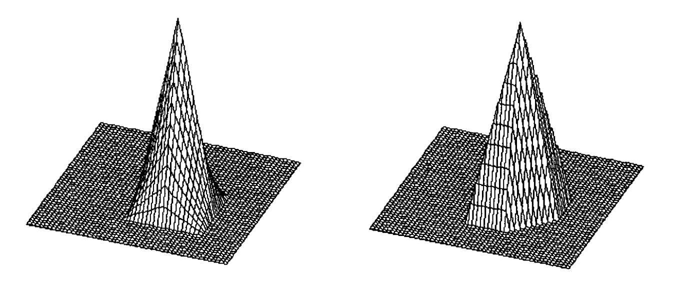


Abbildung 2.9: Darstellung von Ansatzfunktionen

Die Koeffizienten der Gleichung „ziehen" sozusagen an den Ansatzfunktionen und bilden so durch unterschiedliche Werte die Lösung ab. Weitere Beispiele für die Definition von Ansatzfunktionen und ihre analytische Untersuchung finden sich in jedem Lehrbuch über Finite Elemente, also insbesondere in [33, 16].

2.2.4 Berechnung der Integrale

Eine wichtige Teilaufgabe ist die Berechnung der in der schwachen Formulierung auftretenden Integrale $\int_\Omega \phi_i \phi_j \, dx$. Da diese Integrale im allgemeinen nicht analytisch gelöst werden können, ist man hier auch wieder auf numerische Verfahren angewiesen. Diese Verfahren zur numerischen Integralberechnung nennt man Quadraturverfahren. Ein allgemeines Quadraturverfahren sieht folgendermaßen aus:

$$\int_a^b f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i) \text{ mit } x_i \in [a,b].$$

Die x_i heißen Stützstellen, die w_i Gewichte der Quadraturformel. Das Verhalten und die Qualität der Quadraturformel wird von der Wahl dieser Werte maßgeblich beeinflußt. Zwei Verfahren sollen im folgenden vorgestellt werden.
Das erste Verfahren ist das Trapezverfahren. Das Intervall, über das zu integrieren ist, wird in gleich große Abschnitte eingeteilt, und die so entstandenen Trapezflächen werden aufsummiert. Abbildung 2.10 verdeutlicht das Verfahren.

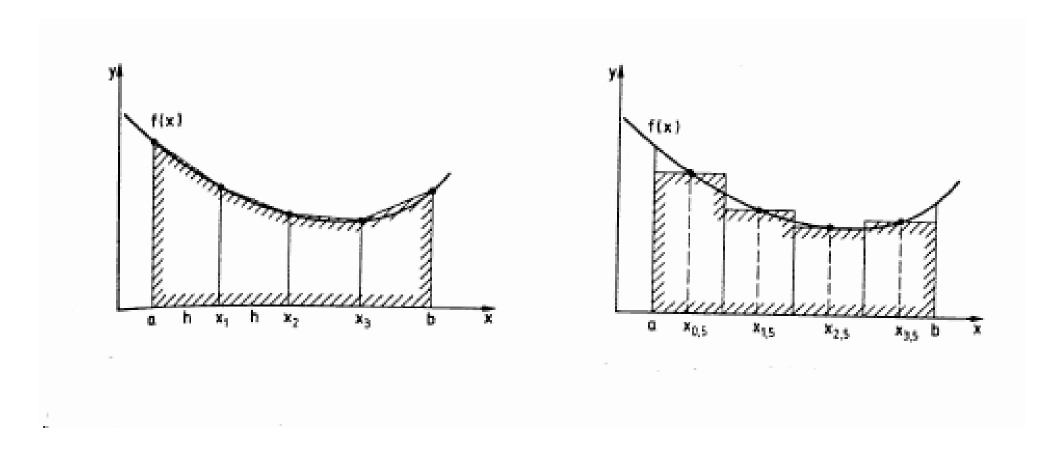


Abbildung 2.10: Trapez- und Mittelpunktverfahren

Die Stützstellen sind durch die Vorschrift $x_j := a + jh$ mit $h = (b-a)/n$ gegeben, d.h. das Intervall wird äquidistant unterteilt. Die Gewichte sind durch $w_0 = 0.5, w_1 = 1, \ldots, w_{n-1} = 1, w_n = 0.5$. definiert.
Die Einzelflächen ergeben sich folgendermaßen:

$$
\begin{aligned}
F_1 &= f(x_1)h + 1/2(f(x_0) - f(x_1))h, \\
F_2 &= f(x_2)h + 1/2(f(x_1) - f(x_2))h, \\
&\text{usw.}
\end{aligned}
$$

Die so gewichtete Summe ergibt eine Approximation des Integrals.
Eine Variation benutzt nicht die Funktionswerte an den Teilintervallgrenzen, sondern in den Mittelpunkten. Dieses Verfahren heißt Mittelpunktverfahren.
Das zweite Verfahren gehört zu der Klasse der sogenannten Interpolationsquadraturformeln. Die grundlegende Idee hinter diesen Verfahren ist die, die zu integrierende Funktion durch ihr Interpolationspolynom zu ersetzen und dieses Polynom dann exakt zu integrieren. Ein Vertreter dieser Klasse ist das Gauß-Verfahren, welches die Nullstellen der sogenannten Legendre-Polynome als Stützstellen benutzt.
Das Legendre-Polynom 11. Ordnung hat folgende Gestalt:

$$
l_{11}(x) = \frac{88179}{256}x^{11} - \frac{230945}{256}x^9 + \frac{109395}{128}x^7 - \frac{45045}{128}x^5 + \frac{15015}{256}x^3 - \frac{693}{256}x.
$$

Die Gauß-Formel ist unter allen anderen Quadraturformeln dadurch ausgezeichnet, daß sie bei n gegebenen Stützstellen noch Polynome des Grades $2n-1$ exakt integrieren kann.
Die Gewichte lassen sich nach folgender Formel ausrechnen:

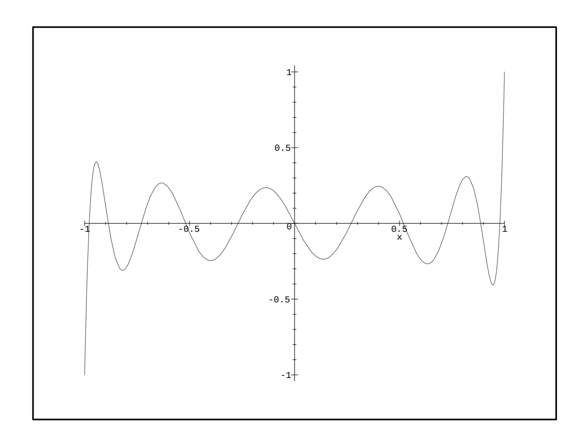


Abbildung 2.11: Legendre-Polynom 11. Ordnung

$$w_k = \int_{-1}^{1} \prod_{\substack{j=1 \\ j \neq k}}^{n} \left(\frac{x - x_j}{x_k - x_j} \right)^2 dx.$$

Diese Ausführungen für den eindimensionalen Fall lassen sich in ähnlicher Weise auf höhere Dimensionen übertragen.

2.2.5 Lösung des Gleichungssystems

Es gibt zwei große Klassen an Verfahren zur Lösung von linearen Gleichungssystemen, die direkten und die iterativen Verfahren. Die direkten Verfahren liefern die Lösung nach einem Schritt, sind dafür aber sehr rechenaufwendig. Die iterativen Verfahren sind pro Schritt weniger rechenintensiv, benötigen aber mehrere, unter Umständen sehr viele Schritte, um die Lösung zu liefern.

Direktes Verfahren: Gauß-Elimination

Es sei ein lineares Gleichungssystem

$$\sum_{k=1}^{n} a_{ik} x_k + b_i = 0, \quad (i = 1, \dots, n)$$

mit n Gleichungen und n Unbekannten zu lösen.
Zur Verdeutlichung des Algorithmus soll folgendes Beispiel mit $n = 4$ dienen:

x_1	x_2	x_3	x_4	1
a_{11}	a_{12}	a_{13}	a_{14}	b_1
a_{21}	a_{22}	a_{23}	a_{24}	b_2
a_{31}	a_{32}	a_{33}	a_{34}	b_3
a_{41}	a_{42}	a_{43}	a_{44}	b_4 .

Auf dieses Schema dürfen die folgenden Operationen angewendet werden, welche das gegebene Gleichungssystem nicht verändern:

- Vertauschen von Zeilen,
- Multiplikation einer ganzen Zeile mit einer Zahl ungleich Null und
- Addition eines Vielfachen einer Zeile zu einer anderen.

Unter der Annahme $a_{11} \neq 0$ (andernfalls wird ein Zeilentausch nach unten durchgeführt) wird von den i-ten ($i \geq 2$) Zeilen das (a_{i1}/a_{11})-fache der ersten subtrahiert und führt zu:

x_1	x_2	x_3	x_4	1
a_{11}	a_{12}	a_{13}	a_{14}	b_1
0	$a_{22}^{(1)}$	$a_{23}^{(1)}$	$a_{24}^{(1)}$	$b_2^{(1)}$
0	$a_{32}^{(1)}$	$a_{33}^{(1)}$	$a_{34}^{(1)}$	$b_3^{(1)}$
0	$a_{42}^{(1)}$	$a_{43}^{(1)}$	$a_{44}^{(1)}$	$b_4^{(1)}$.

Mit den Quotienten $l_{i1} = a_{i1}/a_{11}$ sind die Elemente gegeben durch

$$\begin{aligned} a_{ik}^{(1)} &= a_{ik} - l_{i1}a_{1k}, \\ b_i^{(1)} &= b_i - l_{i1}b_1. \end{aligned}$$

Unter der Annahme $a_{22} \neq 0$ folgt der zweite Eliminationsschritt

x_1	x_2	x_3	x_4	1
a_{11}	a_{12}	a_{13}	a_{14}	b_1
0	$a_{22}^{(1)}$	$a_{23}^{(1)}$	$a_{24}^{(1)}$	$b_2^{(1)}$
0	0	$a_{33}^{(2)}$	$a_{34}^{(2)}$	$b_3^{(2)}$
0	0	$a_{43}^{(2)}$	$a_{44}^{(2)}$	$b_4^{(2)}$.

Nach dem dritten Schritt erhält man folgendes System

x_1	x_2	x_3	x_4	1
a_{11}	a_{12}	a_{13}	a_{14}	b_1
0	$a_{22}^{(1)}$	$a_{23}^{(1)}$	$a_{24}^{(1)}$	$b_2^{(1)}$
0	0	$a_{33}^{(2)}$	$a_{34}^{(2)}$	$b_3^{(2)}$
0	0	0	$a_{44}^{(3)}$	$b_4^{(3)}$.

Wenn man die letzte Zeile mit dem Unbekanntenvektor multipliziert, erhält man die letzte Lösungskomponente. Setzt man diese in die vorletzte Zeile ein und löst nach der vorletzten Unbekanntenkomponente auf, erhält man die vorletzte Lösungskomponente. Dieses Rückwärtseinsetzen führt man für alle Zeilen durch und erhält so die Lösung, allgemein

$$x_i = -\Big[\sum_{k=i+1}^{n} a_{ik}^{(i-1)} x_k + b_i^{(i-1)} \Big] / a_{ii}^{(i-1)}.$$

Dieses Verfahren besitzt ein kubisches Laufzeitverhalten, für n Gleichungen müssen $O(n^3)$ Operationen ausgeführt werden. Dies ist für kleine Problemgrößen akzeptabel, aber nicht für solche, die in komplexen Simulationen auftreten.

Iterative Verfahren

Die zweite Klasse bilden die iterativen Verfahren, die das endgültige Resultat nicht schon nach einem Schritt ausgeben, sondern erst nach mehreren. Dafür sind die Einzelschritte deutlich billiger. Das Kriterium, um die Qualität des Iterationsverhalten zu bewerten, heißt Konvergenzrate ρ und ist folgendermaßen definiert (dabei bezeichne $\mathbf{x}^{(k)}$ das Ergebnis nach k Iteration und $\mathbf{x}^{(0)}$ den Startwert):

$$\rho = \left(\frac{||A\mathbf{x}^{(k)} - \mathbf{b}||}{||A\mathbf{x}^{(0)} - \mathbf{b}||} \right)^{1/k}.$$

Konvergenzraten größer 0.5 sind „schlecht", kleiner als 0.1 sind „gut".

CG-Verfahren Um definitive Aussagen über dieses Verfahren treffen zu können, muß die Koeffizientenmatrix A positiv definit sein.
Eine Matrix $A \in \mathbb{R}^{N \times N}$ heißt positiv definit, wenn gilt:

$$\begin{aligned} (\mathbf{x}, A\mathbf{x}) &> 0 \quad \text{für alle } \mathbf{x} \in \mathbb{R}^N \setminus \{\mathbf{0}\}, \\ (\mathbf{x}, A\mathbf{x}) &= 0 \quad \text{nur für } \mathbf{x} = \mathbf{0}. \end{aligned}$$

Der zur Auflösung erforderliche Aufwand läßt sich durch die Verwendung einer der Aufgabe angepaßten Basis $\{\mathbf{p}_j\}$ des $\mathbb{R}^N$ gezielt reduzieren. Es besitze $\mathbf{p}_j$ die Eigenschaft

$$(A\mathbf{p}_i, \mathbf{p}_j) = \delta_{ij},$$

mit dem Kronecker-Symbol δ_{ij}. Richtungen bzw. Vektoren $\mathbf{p}_j$ mit diesen Eigenschaften heißen konjugiert oder A-orthogonal. Stellt man die gesuchte Lösung $\mathbf{u}$ des Gleichungssystem über der Basis $\{\mathbf{p}_j\}$ dar, d.h.

$$\mathbf{u} = \sum_{j=1}^{N} \eta_j \mathbf{p}_j,$$

dann lassen sich die zugehörigen Koeffizienten $\eta_j, j = 1, \ldots, N$ wegen der A-Orthogonalität explizit darstellen durch

$$\eta_j = \frac{(\mathbf{b}, \mathbf{p}_j)}{(A\mathbf{p}_j, \mathbf{p}_j)}, \quad j = 1, \ldots, N.$$

Konjugierte Richtungen sind in der Regel nicht a-priori bekannt. Der Grundgedanke der CG-Verfahren besteht darin, diese Richtungen mit Hilfe eines Orthogonalisierungsverfahrens aus den verbleibenden Defekten $\mathbf{d}^{(k+1)} := \mathbf{b} - A\mathbf{u}^{(k+1)}$ in den Iterationspunkten $\mathbf{u}^{(k+1)}$ rekursiv zu erzeugen. Man stellt also die neue Richtung $\mathbf{p}^{(k+1)}$ in der Form

$$\mathbf{p}^{(k+1)} = \mathbf{d}^{(k+1)} + \sum_{j=1}^{k} \beta_{kj} \mathbf{p}^{(j)}$$

dar, und bestimmt die Koeffizienten $\beta_{kj} \in \mathbb{R}$ aus der verallgemeinerten Orthogonalitätsbedingung $(A\mathbf{p}^{(k+1)}, \mathbf{p}_j) = 0, j = 1, \ldots, k$.

Die Richtungen $\mathbf{d}^{(k+1)}$, die auch Anti-Gradient der dem linearen Gleichungssystem zugeordneten Funktion

$$F(\mathbf{u}) = 1/2(A\mathbf{u}, \mathbf{u}) - (\mathbf{b}, \mathbf{u})$$

in den Punkten $\mathbf{u}^{(k+1)}$ sind, gaben dem CG-Verfahren (engl. *conjugate gradients*) seinen Namen.

Defektkorrekturverfahren Diese Klasse von Verfahren arbeitet so, daß sie zum Iterationsvektor einen Korrekturwert hinzuaddiert und so näher an die Lösung gelangt.
Zu lösen ist das lineare Gleichungssystem

$$A\mathbf{x} = \mathbf{b},$$

umgeschrieben:

$$\begin{aligned} \mathbf{x} &= A^{-1}\mathbf{b} \\ &= \mathbf{x} - \mathbf{x} + A^{-1}\mathbf{b} \\ &= \mathbf{x} - A^{-1}A\mathbf{x} + A^{-1}\mathbf{b} \\ &= \mathbf{x} - A^{-1}(A\mathbf{x} - \mathbf{b}). \end{aligned}$$

Als Iterationsverfahren formuliert lautet dies im Schritt $k + 1$:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - A^{-1}(A\mathbf{x}^{(k)} - \mathbf{b}).$$

Nun hat man noch nicht so viel gewonnen, denn es bliebe immer noch die Aufgabe A^{-1} zu berechnen. Dies würde wieder Zeit $O(N^3)$ kosten. Für diese sogenannte Vorkonditionierungsmatrix ist es aber nicht notwendig, die kompletten Einträge von A zu nehmen, sondern nur Teile davon, die einfacher zu invertieren sind (Matrix C in der folgenden Gleichung). Nimmt man z.B. nur die Diagonaleinträge von A, sind zur Invertierung nur deren Kehrwerte zu bilden, und man spricht vom Jacobi-Verfahren. Nimmt man die untere Dreiecksmatrix hinzu, die sich durch Rückwärtseinsetzen relativ leicht invertieren läßt, erhält man das sogenannte Gauß-Seidel-Verfahren. Allerdings sind die Konvergenzraten bei diesen Verfahren so schlecht, daß sie als eigenständige Löser in der Praxis nicht verwendet werden. Sie spielen aber eine wichtige Rolle in den noch folgenden Mehrgitterverfahren.
Ferner ist ein sogenannter Relaxationsparameter $\omega < 1$ gebräuchlich, um den Defektkorrekturwert zu dämpfen und somit bessere Konvergenzraten zu erzielen. Die vollständige Vorschrift schreibt sich schließlich:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \omega C^{-1}(A\mathbf{x}^{(k)} - \mathbf{b}).$$

Mehrgitterverfahren Analytische Untersuchungen der oben genannten Defektkorrekturverfahren haben gezeigt, daß die hochfrequenten Anteile des Fehlers gut gedämpft werden, während bei den niederfrequenten Anteilen Schwierigkeiten auftreten und so die langsame Konvergenz verursachen. Eine Idee besteht darin, eine Sequenz von Gittern unterschiedlicher Feinheit

zu verwenden, um auf jedem der Gitter einen oder mehrere Defektkorrekturschritte auszuführen („Glätten”), und die berechnete Näherung für die Korrektur von einem zum nächsten Gitter zu transferieren. Die Operation, um von einem feinen zu einem groben Gitter zu kommen, heißt Restriktion, die umgekehrte Prolongation. Ist man auf dem gröbsten Gitter angelangt, muß die sogenannte Grobgittergleichung exakt gelöst werden. Da diese Probleme im allgemeinen klein sind, werden hier in der Regel direkte Löser benutzt.

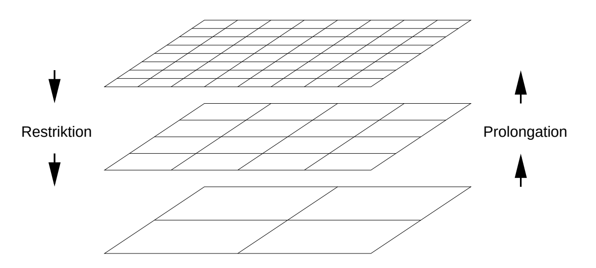


Abbildung 2.12: Mehrgitteroperationen

Mehrgitterverfahren lassen sich durch folgende Parameter spezifizieren (im allgemeinen sind diese Parameter fest für alle Gitterstufen):

- verwendetes Glättungsverfahren,
- Anzahl der Glättungsschritte,
- Wahl der Relaxationsparameter und
- Art und Weise der Gitterstufenwechsel (Zyklus). Hierbei sind drei Zyklen gebräuchlich, V, W und F. Der F-Zyklus folgt dem V-Zyklus beim Abstieg und dem W-Zyklus beim Aufstieg.

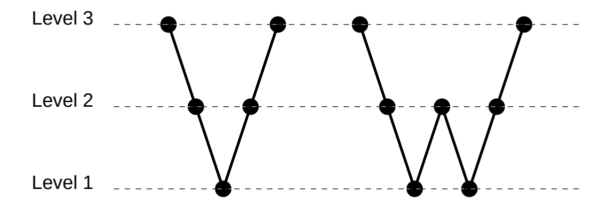


Abbildung 2.13: Mehrgitter-Zyklen

Der Mehrgitteralgorithmus läßt sich also folgendermaßen zusammenfassen:

1. Vorglättung,
2. Berechnung des Korrekturterms durch Abstieg zum Grobgitter,
3. Prolongation der Lösung zum Feingitter und
4. Nachglättung.

Die Mehrgitterverfahren zeichnen sich durch einen linearen Aufwand aus und die Konvergenzraten sind im allgemeinen sehr gut (Konvergenzrate ≈ 0.1). Außerdem ist die Iterationszahl unabhängig von der Gitterweite. Sie bilden sozusagen die Allzweckwaffe.

2.2.6 Zeitschrittverfahren

In komplexen Differentialgleichungen, wie z.B. den Navier-Stokes-Gleichungen für Strömungsprobleme, kommt der Parameter Zeit t vor, während die Gleichung aber auch Informationen über den Ort liefert.
Das gebräuchlichste Vorgehen ist, erst im Ort und dann in der Zeit zu diskretisieren. Die Ortsdiskretisierung kann, wie oben beschrieben, mit der Methode der Finiten Elemente durchgeführt werden. Einsetzen dieser semidiskreten Form in die schwache Formulierung ergibt eine gewöhnliche Differentialgleichung in der Zeit, wobei die bereits diskretisierte Ortsableitung als Anfangswert eingeht (zur Erinnerung: partielle Differentialgleichungen enthalten Ableitungen nach Orts- und Zeitvariablen, gewöhnliche nur nach einer Variablen):

$$\frac{du}{dt} + R(u,t) = 0, \quad u(t_n) = u_n, \text{ mit } R(u,t) := Lu - f.$$

L ist beim Poissonproblem (s. Kap. 2.2.1) der Laplace-Operator, allgemein die linke Seite der Gleichung. Die noch kontinuierliche Zeitableitung $\frac{du}{dt}$ kann nun mit dem Standard-Θ-Schema diskretisiert werden:

$$\frac{u_{n+1} - u_n}{\Delta t} + \Theta R_{n+1} + (1-\Theta) R_n = 0.$$

Dabei ist $\Theta \in [0,1]$ und für $\Theta = 0.5$ ergibt sich das *Crank-Nicolson*-Verfahren der Ordnung 2. Weitere Verfahren finden sich beispielsweise in [33, 16].
Eine Alternative (dies führt auf *Taylor-Galerkin*-Verfahren) diskretisiert zunächst in der Zeit mit einer Taylorentwicklung höherer Ordnung (3-4) und ersetzt dann diese höhere Zeitableitung durch Ortsableitungen:

$$\begin{aligned} u_{n+1} &\approx u_n + \Delta t \frac{\partial u_n}{\partial t} + \frac{(\Delta t)^2}{2} \frac{\partial^2 u_n}{\partial t^2}, \\ u_{n+1} &\approx u_n - \Delta t L u_n + \frac{(\Delta t)^2}{2} L^2 u_n. \end{aligned}$$

Jetzt werden alle verbleibenden Ortsableitungen typischerweise mit der Methode der Finiten Elemente diskretisiert. Die Auswertung der vorkommenden, höheren Ortsableitungen kann durch Unterteilung in mehrere Teilschritte vermieden werden. Verfahren dieses Typs sind deutlich aufwendiger, zeichnen sich aber durch höhere erreichbare Ordnungen in der Zeit und bessere Stabilitätseigenschaften aus.

2.2.7 Anwendbarkeit von „computational steering" Konzepten

Die Idee des *computational steering* (s. Kap. 2.1.3) ist leider nicht generell auf numerische Strömungssimulation übertragbar. Zu unterscheiden ist zwischen Problemen im Ort und in der Zeit. Die theoretische Untersuchung obiger Problemklassen („parabolisch", „hyperbolisch" und „elliptisch") im Ort führt zu den sogenannten *Charakteristiken* als primäres Unterscheidungsmerkmal. Charakteristiken sind – anschaulich – Wellenfronten, die Kurven im Raum folgen und in einem Fluß Gebiete trennen, die schon beeinflußt wurden, von Gebieten, die noch nicht gestört wurden. Parabolische Probleme haben eine Charakteristik, hyperbolische zwei und elliptische keine. Es werden nun Abhängigkeits- und Einflußgebiete von Punkten auf einer Charakteristik untersucht, die vollständige Durchführung dieser Analyse ist in [19] zu finden. Dabei haben Punkte aus der *domain of dependence* einen Punkt P auf einer Charakteristik zu

einem Randpunkt B vorher beeinflußt, und P wirkt selbst auf Punkte aus der *zone of influence*. Die folgende Abbildung zeigt das Ergebnis:

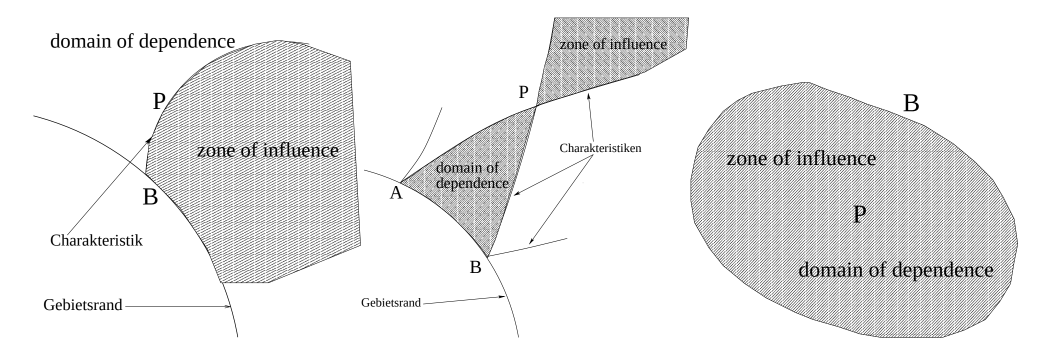


Abbildung 2.14: Einfluß- und Abhängigkeitsgebiete für parabolische, hyperbolische und elliptische Probleme nach [19]

Parabolische Probleme sind demzufolge relativ gut für die Untersuchung in *computational steering*-Umgebungen geeignet, da sie eine klare Trennung in „Vergangenheit”, „Gegenwart” und „Zukunft” aufweisen. Hyperbolische Probleme bieten je nach Verlauf der Charakteristiken schon weniger Freiheit, und elliptische Probleme zeichnen sich durch unendliche Ausbreitungsgeschwindigkeit von Informationen aus – das bedeutet, jeder Punkt beeinflußt jeden anderen sofort – und machen somit die Untersuchung unmöglich.

In der Zeit ist vieles einfacher: Alle instationären Probleme sind parabolisch in der Zeit, d.h. theoretisch für die Verwendung innerhalb von *computational steering*-Umgebungen geeignet. Auch stationäre Probleme können durch eine Technik namens *quasi-timestepping* instationär gelöst und somit gleich behandelt werden.

Abschließend sei bemerkt, daß dies nur ein kurzer Abriß der theoretischen Möglichkeiten ist. In der Praxis gibt es sehr viele Charakteristiken, weil es sehr viele Gitterpunkte gibt. Bei klar ausgeprägten Flüssen kann versucht werden, passende Gebiete außerhalb der Einflußgebiete zu finden, sonst wird man die Verwendung von Interpolationstechniken nach der Parameteränderung vor dem Wiederaufsetzen der Simulation nicht umgehen können. Interpolation führt aber leider zu neuen Fehlern und so kann es schlimmstenfalls passieren, daß die Beobachtung der Simulation in einer *computational steering*-Umgebung die Simulation bzw. ihr Ergebnis verfälscht.

2.3 Wissenschaftliches Rechnen

Wissenschaftliches Rechnen beschreibt die konkrete Umsetzung der im vorherigen Kapitel beschriebenen Verfahren in Algorithmen und ihre hocheffiziente Implementierung auf Supercomputern. Wissenschaftliches Rechnen nimmt somit einen Platz im Grenzgebiet zwischen praktischer Informatik und angewandter Mathematik ein.

Auf der anderen Seite wird mit dem Begriff des wissenschaftlichen Rechnens auch die konkrete Durchführung von Simulationen von Seiten der Anwender bezeichnet.

2.3.1 Hocheffiziente Implementierung

Die oben angesprochene Diskretisierung der partiellen Differentialgleichungen mit Finiten Elementen schafft viele Probleme: Aus Gründen der numerischen Stabilität und der geforderten Genauigkeit müssen hochauflösende Verfahren verwendet werden, das bedeutet, es wird mit einem sehr feinen Ortsgitter und nahe bei Null liegenden Zeitschrittweiten gerechnet. Als Konsequenz sind Problemgrößen von einer Million und mehr Unbekannten keine Seltenheit. Direkte Verfahren scheiden also wegen ihres kubischen Laufzeitverhaltens aus.

Auf handelsüblichen PCs resultieren selbst hocheffiziente Lösungsstrategien wie das CG - Verfahren (s. Kap. 2.2.5) in unzumutbar langen Berechnungszeiten. Ein Ausweg ist die Portierung auf andere Rechnerarchitekturen, die so genannten Supercomputer. Zwei verschiedene Modelle lassen sich voneinander trennen:

- Vektorrechner arbeiten nicht auf skalaren Variablen, sondern können Befehle direkt auf einen ganzen Vektor komponentenweise anwenden. Klar ist: bei allen komponentenweise definierten Vektoroperationen wie der Addition ist diese Architektur wesentlich schneller, unter Vernachlässigung des Mehraufwands zur Verwaltung genau um die Anzahl der gleichzeitig verarbeitbaren Elemente. Bei Operationen, die mehrere Komponenten desselben Vektors verknüpfen, sind Vektorrechner nicht schneller als normale, skalare Architekturen.

  Ein Beispiel soll dies verdeutlichen, wobei o.B.d.A. die Breite der Vektoreinheit für die hier verwendete Vektorlänge n ausreicht. Berechnet werden soll die euklidische Norm eines Vektors $\mathbf{a} \in \mathbb{R}^n$:

  $$||\mathbf{a}||_2 := \sqrt{\sum_{i=1}^{n} a_i^2}$$

  Die innerste Operation ist einfach vektorisierbar, es wird der Vektor $\mathbf{a}$ als doppelte Eingabe für die komponentenweise Multiplikation verwendet. Die anschließende Addition der einzelnen Komponenten kann auf einem Vektorrechner nicht schneller durchgeführt werden, und das Ziehen der Wurzel ist eine rein skalare Operation.

- Massivparallele Maschinen zeichnen sich durch eine große Anzahl an Prozessoren aus. Für den Speicherzugriff gibt es zwei verschiedene Modelle: *shared memory*- und *distributed memory*-Architekturen: erstere greifen auf denselben Datenbestand zu, bei letzteren hat jeder Prozessor einen eigenen Speicher, die einzelnen Prozessoren arbeiten also auf verschiedenen Daten. *Distributed-memory*-Architekturen lassen sich je nach der Verbindung der einzelnen Prozessoren oder Knoten weiter unterteilen: Cluster sind zu einem massiv parallelen Computer zusammengeschaltete, normale PCs oder Workstations. Die Verbindung ist hier durch ein Computernetzwerk realisiert. Die zweite Möglichkeit ist die schnelle interne Bus-Verbindung der Knoten.

Allen diesen Architekturen ist gemeinsam, daß die Portierung von Programmen für PCs in der Regel nicht automatisch durch Compiler geschehen kann: Die Programmierung der Maschinentypen unterscheidet sich zu stark voneinander.

2.3.2 Ein Beispiel für Algorithmen auf Vektorrechnern

Bei Vektormaschinen müssen die vorkommenden Operationen "vektorisiert" werden. Anhand eines Beispiels – die sogenannte "zyklische Reduktion" – soll hier verdeutlicht werden, wie aufwendig dies ist. Die Details sind [33] und einer unveröffentlichten Seminararbeit eines Teilnehmers der Projektgruppe entnommen.

Die zyklische Reduktion ist eine spezielle Implementierung des Thomas-Algorithmus – eine Modifikation des Gauß-Algorithmus (s. Kap. 2.2.5) – für die Lösung tridiagonaler Gleichungssysteme auf Vektorrechnern. Gegeben sei das trilineare System

$$\underbrace{\begin{pmatrix} a_1 & b_1 & 0 & \cdots & \cdots & 0 \\ c_2 & a_2 & b_2 & \ddots & & \vdots \\ 0 & c_3 & a_3 & b_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & b_{n-1} \\ 0 & \cdots & \cdots & 0 & c_n & a_n \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{pmatrix}}_{\mathbf{x}} + \underbrace{\begin{pmatrix} d_1 \\ \vdots \\ \vdots \\ d_n \end{pmatrix}}_{\mathbf{d}} = 0. \tag{2.8}$$

Sei $n = 2m$ und $i \in \mathbb{N}$ gerade mit $1 < i < n$. Betrachtet werden die folgenden drei benachbarten Gleichungen aus Gleichung 2.8:

$$\begin{array}{ccccccccccc} c_{i-1}x_{i-2} & + & a_{i-1}x_{i-1} & + & b_{i-1}x_i & & & & & + & d_{i-1} & = & 0, \\ & & c_i x_{i-1} & + & a_i x_i & + & b_i x_{i+1} & & & + & d_i & = & 0, \\ & & & & c_{i+1}x_i & + & a_{i+1}x_{i+1} & + & b_{i+1}x_{i+2} & + & d_{i+1} & = & 0. \end{array}$$

Durch Addition des $-c_i/a_{i-1}$-fachen der ersten zur mittleren Gleichung und durch Addition des $-b_i/a_{i+1}$-fachen der dritten zur mittleren Gleichung können die Unbekannten x_{i-1} und x_{i+1} eliminiert werden:

$$-\frac{c_{i-1}c_i}{a_{i-1}}x_{i-2} + \left(-\frac{b_{i-1}c_i}{a_{i-1}} + a_i - \frac{b_i c_{i+1}}{a_{i+1}}\right)x_i - \frac{b_i b_{i+1}}{a_{i+1}}x_{i+2} + \left(-\frac{d_{i-1}c_i}{a_{i-1}} + d_i - \frac{b_i d_{i+1}}{a_{i+1}}\right) = 0$$

Damit diese Formel für alle $i = 2, 4, \ldots, n$ gilt, definiert man als Erleichterung zusätzlich:

$$c_i := c_{n+1} := b_n := b_{n+1} := d_{n+1} := 0 \qquad \text{sowie} \qquad a_{n+1} := 1.$$

Jetzt können im *Reduktionsschritt* Koeffizienten für die neuen Gleichungen der geraden Indexwerte i eingeführt werden:

$$\left.\begin{array}{lcl} c_i^{(1)} & := & -c_{i-1}c_i/a_{i-1} \\ a_i^{(1)} & := & -b_{i-1}c_i/a_{i-1} + a_i - b_i c_{i+1}/a_{i+1} \\ b_i^{(1)} & := & -b_i b_{i+1}/a_{i+1} \\ d_i^{(1)} & := & -d_{i-1}c_i/a_{i-1} + d_i - b_i d_{i+1}/a_{i+1} \end{array}\right\} \quad i = 2, 4, \ldots, n \tag{2.9}$$

Damit erhält das Gleichungssystem folgende Gestalt:

$$\begin{pmatrix} a_1 & b_1 & & & & & & & & & \\ 0 & a_2^{(1)} & 0 & b_2^{(1)} & & & & & & & \\ & c_3 & a_3 & b_3 & & & & & & & \\ & c_4^{(1)} & 0 & a_4^{(1)} & 0 & b_4^{(1)} & & & & & \\ & & & c_5 & a_5 & b_5 & & & & & \\ & & & c_6^{(1)} & 0 & a_6^{(1)} & 0 & b_6^{(1)} & & & \\ & & & & & & & \ddots & & & \\ & & & & & & & c_n^{(1)} & 0 & a_n^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ \\ \vdots \\ \\ \vdots \\ \\ x_n \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2^{(1)} \\ d_3 \\ d_4^{(1)} \\ d_5 \\ d_6^{(1)} \\ \vdots \\ d_n^{(1)} \end{pmatrix} = 0. \quad (2.10)$$

Man sieht, daß zum einen die Gleichungen mit geraden Indizes ein lineares Gleichungssystem der Ordnung m mit tridiagonaler Koeffizientenmatrix für die Unbekannten mit geradem Index bilden, und sich zum anderen die Unbekannten mit ungeraden Indizes durch Vorwärtseinsetzen bei bereits berechneten Werten der Unbekannten mit geraden Indizes aus den Gleichungen mit ungeraden Indizes berechnen lassen. Hat nun das durch zyklische Reduktion gewonnene Gleichungssystem halber Ordnung wieder eine gerade Anzahl an Unbekannten, kann der Prozeß rekursiv fortgesetzt werden.
Dieser modifizierte Algorithmus, konkret der Reduktionsschritt (2.9), läßt sich vektorisieren: Zunächst spaltet man die Koeffizientenvektoren $\mathbf{c}$, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{d} \in \mathbb{R}^n$ bezüglich der ungeraden ($^{(u)}$) und geraden ($^{(g)}$) Indizes wie folgt auf:

$$\mathbf{c}^{(\mathbf{u})} := \begin{pmatrix} c_1 \\ c_3 \\ c_5 \\ \vdots \\ c_{n+1} \end{pmatrix}, \mathbf{a}^{(\mathbf{u})} := \begin{pmatrix} a_1 \\ a_3 \\ a_5 \\ \vdots \\ a_{n+1} \end{pmatrix}, \mathbf{b}^{(\mathbf{u})} := \begin{pmatrix} b_1 \\ b_3 \\ b_5 \\ \vdots \\ b_{n+1} \end{pmatrix}, \mathbf{d}^{(\mathbf{u})} := \begin{pmatrix} d_1 \\ d_3 \\ d_5 \\ \vdots \\ d_{n+1} \end{pmatrix} \in \mathbb{R}^{m+1},$$

$$\mathbf{c}^{(\mathbf{g})} := \begin{pmatrix} c_2 \\ c_4 \\ c_6 \\ \vdots \\ c_n \end{pmatrix}, \mathbf{a}^{(\mathbf{g})} := \begin{pmatrix} a_2 \\ a_4 \\ a_6 \\ \vdots \\ a_n \end{pmatrix}, \mathbf{b}^{(\mathbf{g})} := \begin{pmatrix} b_2 \\ b_4 \\ b_6 \\ \vdots \\ b_n \end{pmatrix}, \mathbf{d}^{(\mathbf{g})} := \begin{pmatrix} d_2 \\ d_4 \\ d_6 \\ \vdots \\ d_n \end{pmatrix} \in \mathbb{R}^{m}.$$

Im weiteren werden die folgende Vektoroperationen benötigt:

- $\oplus$ bezeichne die komponentenweise Addition zweier Vektoren: $(\mathbf{x} \oplus \mathbf{y})_i := x_i + y_i$,
- $\otimes$ die komponentenweise Multiplikation: $(\mathbf{x} \otimes \mathbf{y})_i := x_i y_i$,
- $_{+1}$ die Vernachlässigung des ersten Indexwertes:
  $(x_1, x_2, x_3, \ldots, x_n)^T_{+1} := (x_2, x_3, \ldots, x_n)^T$,
- $_{-1}$ die Vernachlässigung des letzten Indexwertes:
  $(x_1, x_2, \ldots, x_{n-1}, x_n)^T_{-1} := (x_1, x_2, \ldots, x_{n-1})^T$.

Man beachte, daß diese Operationen auf einem Vektorrechner in nur einem Schritt durchgeführt werden können.
Nun werden Hilfsvektoren gebildet:

$$\mathbf{r} := \begin{pmatrix} -1/a_1 \\ -1/a_3 \\ \vdots \\ -1/a_{n+1} \end{pmatrix} \in \mathbb{R}^{m+1}; \qquad \mathbf{p} := \mathbf{c^{(g)}} \otimes \mathbf{r_{-1}} = \begin{pmatrix} -c_2/a_1 \\ -c_4/a_3 \\ \vdots \\ -c_n/a_{n-1} \end{pmatrix} \in \mathbb{R}^m;$$

$$\mathbf{q} := \mathbf{b^{(g)}} \otimes \mathbf{r_{+1}} = \begin{pmatrix} -b_2/a_3 \\ -b_4/a_5 \\ \vdots \\ -b_n/a_{n+1} \end{pmatrix} \in \mathbb{R}^m.$$

Damit lassen sich die gesuchten Vektoren des Reduktionsschrittes (2.9)

$$\mathbf{c^{(1)}} := \begin{pmatrix} c_2^{(1)} \\ c_4^{(1)} \\ \vdots \\ c_n^{(1)} \end{pmatrix}, \quad \mathbf{a^{(1)}} := \begin{pmatrix} a_2^{(1)} \\ a_4^{(1)} \\ \vdots \\ a_n^{(1)} \end{pmatrix}, \quad \mathbf{b^{(1)}} := \begin{pmatrix} b_2^{(1)} \\ b_4^{(1)} \\ \vdots \\ b_n^{(1)} \end{pmatrix}, \quad \mathbf{d^{(1)}} := \begin{pmatrix} d_2^{(1)} \\ d_4^{(1)} \\ \vdots \\ d_n^{(1)} \end{pmatrix} \quad \in \mathbb{R}^m$$

folgendermaßen durch reine Vektoroperationen bestimmen, indem einfach die Gleichungen des Reduktionsschrittes (2.9) in die neue Schreibweise übertragen werden:

$$\begin{array}{lllllllll} \mathbf{c^{(1)}} & = & \mathbf{c^{(u)}} & \otimes & \mathbf{p} & & & & \\ \mathbf{a^{(1)}} & = & \mathbf{a^{(g)}} & \oplus & (\mathbf{b^{(u)}} & \otimes & \mathbf{p}) & \oplus & (\mathbf{c^{(u)}_{+1}} \otimes \mathbf{q}) \\ \mathbf{b^{(1)}} & = & \mathbf{b^{(u)}_{+1}} & \otimes & \mathbf{q} & & & & \\ \mathbf{d^{(1)}} & = & \mathbf{d^{(g)}} & \oplus & (\mathbf{d^{(u)}} & \otimes & \mathbf{p}) & \oplus & (\mathbf{d^{(u)}_{+1}} \otimes \mathbf{q}) \end{array} \tag{2.11}$$

Auch der Schritt des Rückwärtseinsetzens läßt sich mittels der speziellen Vektoroperationen schreiben. Dazu verwendet man die folgenden zwei erweiterten Vektoren:

$$\mathbf{x^{(g)}} := (x_0, x_2, x_4, \ldots, x_n, x_{n+2} = 0)^T \in \mathbb{R}^{m+2},$$

$$\mathbf{x^{(u)}} := (x_1, x_3, x_5, \ldots, x_{n-1}, x_{n+1} = 0)^T \in \mathbb{R}^{m+1}.$$

Damit lautet das Rückwärtseinsetzen:

$$\mathbf{x^{(u)}} = (\mathbf{c^{(u)}} \otimes \mathbf{x^{(g)}} \oplus \mathbf{b^{(u)}} \otimes \mathbf{x^{(g)}_{+1}} \oplus \mathbf{d^{(u)}}) \otimes \mathbf{r}. \tag{2.12}$$

Der Thomas-Algorithmus benötigt auf skalaren wie vektoriellen Architekturen $6n$ arithmetische Operationen.
Wie man an den Vektorgleichungen (2.11) und an der Deklaration der Hilfsvektoren sieht, werden pro Reduktionsschritt etwa m Divisionen, $8m$ Multiplikationen und $4m$ Additionen benötigt, insgesamt also $13m$ arithmetische Operationen. Aus (2.12) liest man ab, daß das Rückwärtseinsetzen etwa $3(m+1)$ Multiplikationen und $2(m+1)$ Additionen kostet.
Damit erfordert jeder Reduktionsschritt insgesamt einen Rechenaufwand vom $18m$ Operationen.

Falls $n = 2^q$, $q \in \mathbb{N}$, so kann die zyklische Reduktion genau q mal durchgeführt werden (das im q-ten Schritt zu lösende Gleichungssystem besteht nur aus einer Gleichung in einer Unbekannte und kann trivial gelöst werden). Da jeder Reduktionsschritt und jedes Rückwärtseinsetzen den halben Rechenaufwand im Vergleich zum vorherigen benötigt, ergibt die Summation in diesem Fall einen Rechenaufwand von $18n$ arithmetischen Operationen. Die Methode der zyklischen Reduktion ist also auf skalaren Rechnern in etwa dreimal so teuer wie der Thomas-Algorithmus.
Allerdings liegt die Ausführungszeit auf einem Vektorrechner deutlich darunter. Zunächst wird angenommen, ein ”idealer” Vektorrechner mit einer Vektorlänge von n existiere. In diesem Fall belaufen sich die Kosten auf eine Skalar - Vektor - Multiplikation und zwei Vektor - Vektor - Multiplikationen ($\otimes$) zur Aufstellung der Hilfsvektoren, auf weitere 6 Vektor - Vektor - Multiplikationen und 4 Vektor - Vektor - Additionen im Reduktionsschritt, und noch mal drei Vektor - Vektor - Multiplikationen und zwei Vektor - Vektor - Additionen beim Rückwärtseinsetzen, insgesamt also Kosten von 18 Vektoroperationen. Falls wieder $n = 2^q$ gilt, so erfordert die Lösung des Tridiagonalsystems durch zyklische Reduktion nur noch $18q$ statt $18n$ Operationen, bzw., da $q = \log_2 n$, asymptotisch logarithmische Kosten.
In realistischen Szenarien kommt noch ein logarithmischer Faktor für die Kombination einzeln gelöster Teilsysteme hinzu.
Dieses Beispiel hat gezeigt, daß die Vektorisierung hochgradig nichttrivial ist und bewirken kann, daß ein Programm nach der Anpassung auf anderen Architekturen deutlich an Performanz einbüßt.

2.3.3 Parallelarchitekturen

Nach diesem ausführlichen Exkurs in die Welt der Vektorrechner werden im folgenden einige Details bei der Implementierung auf parallelen Rechnern angedeutet:
Steht eine schnelle Verbindung zwischen den Knoten zur Verfügung, können Algorithmen entworfen werden, die intensiv miteinander kommunizieren dürfen. Auf Clustern, die eventuell nur über Ethernet verbunden sind – im Extremfall nur über eine Modem-Verbindung, wie dies beim *ubiquitous computing*, beispielsweise beim SETI-Programm der NASA, der Fall ist – müssen die Algorithmen grundsätzlich anders entworfen werden, so daß die einzelnen Knoten unabhängig voneinander arbeiten können.
Auch die Ausnutzung maschinenabhängiger Gegebenheiten wie der Cachegröße kann enorme Geschwindigkeitsgewinne oder -einbußen nach sich ziehen. Da die Transfergeschwindigkeit zwischen Cache und RAM um ganze Größenordnungen differiert, und da ein komplettes Gitter nie in den Cache paßt, kann schon eine Änderung der Element- und Knotennummerierungen gewaltige Laufzeitunterschiede nach sich ziehen. Feste Regeln gibt es hier allerdings nicht.

2.4 Wissenschaftliche Visualisierung von Simulationsdaten

2.4.1 Die Visualisierungspipeline

Die Visualisierungspipeline beschreibt den Prozeß, Daten bzw. Informationen in Bilder umzuwandeln bzw. zu kodieren. Dazu werden folgende Stufen durchlaufen: Als erstes wird eine Datentransformation (engl. *filtering*) vorgenommen, in der die Ausgangsdaten oder Rohdaten aufbereitet werden, z.B. können die Daten durch Schwellwertoperationen, Interpolation oder

Rundung reduziert, vervollständigt oder konvertiert werden. Anschließend werden die aufbereiteten Daten in der Visualisierungstransformation (engl. *mapping*) in Geometriedaten umgewandelt. Im letzten Schritt dann werden die Geometriedaten in der visuellen Abbildungstransformation (engl. *rendering*) in Bilddaten transformiert, die auf einem Bildschirm ausgegeben werden können.

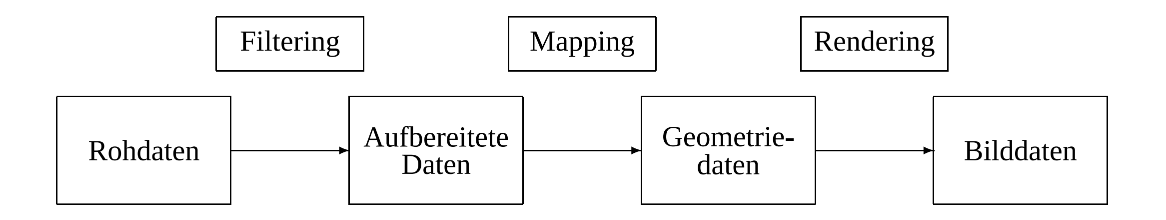


Abbildung 2.15: Die Visualisierungspipeline

2.4.2 Klassifizierung von Visualisierungstechniken

Im folgenden werden verschiedene Ansätze zur Klassifikation von Visualisierungstechniken diskutiert. Sehr einfach einzusehen ist die Einteilung nach der Dimension des Definitionsgebiets: Algorithmen arbeiten auf ein-, zwei- oder dreidimensionalen Daten. Diese Klassifikation ist einfach, aber nicht sehr weitreichend, da sehr viele, sehr unterschiedliche Algorithmen in dieselbe große Klasse eingeordnet werden. Ein anderer Ansatz ordnet die Techniken nach der Dimension des Lösungsraumes: Simulationen können nulldimensionale Daten (also Punktwolken), skalare Daten oder vektorielle Daten liefern. Diese Methode ist mächtiger, vernachlässigt aber die Komplexitätssteigerung bei wachsender Gitterdimension. Eine dritte Möglichkeit ist die Einteilung gemäß der Gittertypen (s. Kap. 2.7), dies ist aber eine rein algorithmische und keine konzeptionelle Unterscheidung, da beispielsweise Methoden zur Zellsuche auf kartesischen Gittern wesentlich einfacher zu implementieren sind als auf beliebig krummberandeten Gittern. Als Erweiterung von [6] wird in der Projektgruppe eine Mischform bzw. mehrstufige Klassifikation verwendet: Die primäre Unterscheidung erfolgt nach den Dimensionen von Definitions- und Lösungsraum. Nur bei Bedarf wird eine Klasse weiter eingeteilt gemäß der Gittergeometrie. Etwas konkreter werden zum einen Punktwolken, zum andern die Funktionenklassen $\mathbb{R} \to \mathbb{R}$, $\mathbb{R}^2 \to \mathbb{R}$, $\mathbb{R}^3 \to \mathbb{R}$, $\mathbb{R}^2 \to \mathbb{R}^2$, $\mathbb{R}^2 \to \mathbb{R}^3$, $\mathbb{R}^3 \to \mathbb{R}^3$ unterschieden.

dim / dim	Punktwolke	skalar	2D - vektoriell	3D - vektoriell
1D	1D - Scatter Plot	Liniendiagramm Tortendiagramm Histogramm	-	-
2D	2D - Scatter Plot	Isolinien Schattierte Plots Bilder Gitternetze Schattierte Flächen Höhenfelder	Pfeilplots Strömungslinien Partikelverfolgung Pfadverfolgung LIC Diffusionsmodelle	Pfeilplots
3D	3D - Scatter Plot	Schnittflächen Isoflächen Voxelmodelle	-	Pfeilplots Strömungslinien Partikelverfolgung Pfadverfolgung

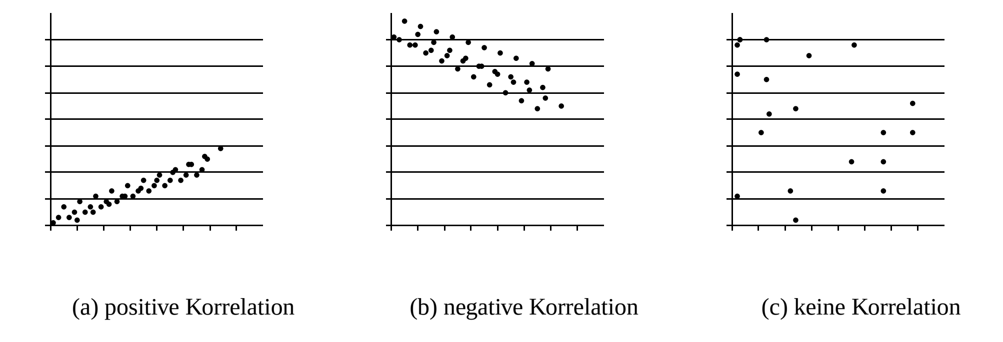
Die Bereiche zwei- und dreidimensionaler skalarer und vektorieller Daten können aus algorithmischer Sicht (s.o.) jetzt noch nach Gittertypen aufgespalten werden.

2.4.3 Techniken für skalare Daten

Alle in diesem Abschnitt betrachteten Funktionen bilden in die reellen Zahlen ab.

Techniken für Scatter Plots

Punkt- bzw. Streudiagramme (engl. *scatter plots*) sind die einfachste Art, große Datenmengen darzustellen. Dabei wird zur Darstellung für jeden Wert ein Symbol (übliche Symbole sind z.B. Punkte, Kreise oder Dreiecke) in das Diagramm eingezeichnet. Durch unterschiedliche Symbole für verschiedene Variablen lassen sich mehrere Diagramme zusammenfassen und so die Übersichtlichkeit evtl. steigern. Dieses Vorgehen ist zunächst dimensionsunabhängig, Daten können auf einer Achse, in einer Ebene (s. Abbildung 2.16) oder in einem Quader dargestellt werden.



(a) positive Korrelation (b) negative Korrelation (c) keine Korrelation

Abbildung 2.16: Verschiedene 2D Streudiagramme

Techniken für eindimensionale Funktionen

Eindimensionale Funktionen sind der wohl einfachste Fall, denn die Daten werden hier von einer einfachen Funktion $x \mapsto f(x)$ bestimmt. Als Techniken zur Darstellung von eindimensionalen Funktionen existieren beispielsweise Linien-, Balken- und Tortendiagramme sowie Histogramme.

Liniendiagramme Bei Liniendiagrammen (engl. *line graphs*) wird die zugrunde liegende Funktion entlang der Werte gezeichnet. Falls die zu zeichnende Funktion nicht direkt bekannt ist, muß interpoliert werden, indem z.B. benachbarte Funktionswerte durch Geradensegmente verbunden werden (s. Abb. 2.17).

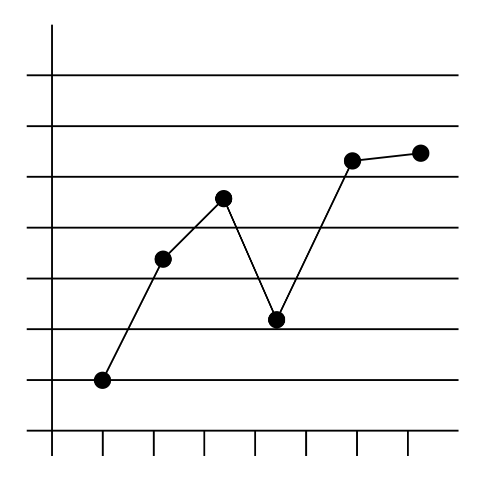


Abbildung 2.17: Liniendiagramm mit linearer Interpolation

Balkendiagramme und Tortendiagramme Balkendiagramme (engl. *bar charts*) stellen Variablen als horizontale oder vertikale Balken dar, deren Werte durch die Länge repräsentiert wird. Eine verwandte Art stellen die Tortendiagramme dar, bei denen Variablen als Tortenstücke eines Ganzen visualisiert werden, die die Relationen zwischen den verschiedenen Werten wiedergeben.

Histogramme Histogramme stellen eine etwas andere Art der Visualisierung dar, denn hier werden die Daten nicht direkt dargestellt, sondern die Häufigkeit von Werten. Dafür wird der Wertebereich der Funktion in Intervalle unterteilt, und dann für alle Intervalle die Anzahl aller Werte, die in dieses Intervall fallen, bestimmt. Die so ermittelten Häufigkeiten werden danach als Balkendiagramm ausgegeben.

Techniken für zweidimensionale Funktionen

Bei den zweidimensionalen Funktionen $(x_1, x_2) \mapsto f(x_1, x_2)$ gibt es zunächst analog zu den eindimensionalen Funktionen die Möglichkeit, Balken- oder Histogramme einzusetzen, da diese sich ohne Probleme erweitern lassen. Weiterhin unterscheidet man zwei Klassen von Funktionen, je nachdem, wie die Funktionswerte vorliegen, ob sie auf einem (wie auch immer strukturierten) Gitter oder beliebig verstreut liegen.

Isolines Isolinien (engl. *isolines*) verbinden im zweidimensionalen Raum alle Punkte, die den gleichen Wert haben, wobei diese Werte Parameter der Darstellung sind. Als Beispiel kann man sich hier Landkarten mit eingetragenen Höhenlinien (s. Abb. 2.19(a)) vorstellen, die nichts anderes als Isolinien sind, da sie alle Punkte der gleichen Höhe miteinander verbinden. Um Isolinien auf einem Gitter zu erstellen, markiert man erst alle Gitterpunkte, je nachdem, ob sie über oder unter dem gesuchten Isowert c liegen, positiv bzw. negativ. Jetzt gibt es zwei unterschiedliche Ansätze, das Problem weiter anzugehen:
Beim Cell-Order-Verfahren geht man Zelle für Zelle im Gitter ab und bestimmt, ob es mindestens einen Vorzeichenwechsel zwischen den Eckpunkten dieser Zelle gibt. Ist dies der Fall, dann schneidet die Isolinie die Zelle mindestens einmal. Im Fall, daß das Vorzeichen viermal wechselt, wird die Zelle sogar von zwei Isoliniensegmenten geschnitten und es existieren zwei Möglichkeiten, wie diese Segmente durch die Zelle verlaufen (s. Abb. 2.18).

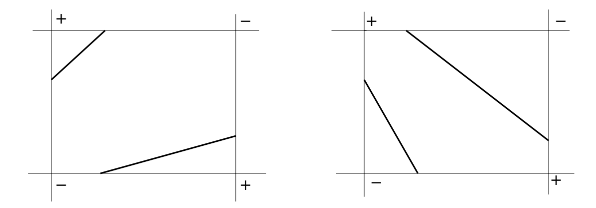


Abbildung 2.18: Zwei Möglichkeiten, wie Isolinien durch eine Zelle verlaufen können

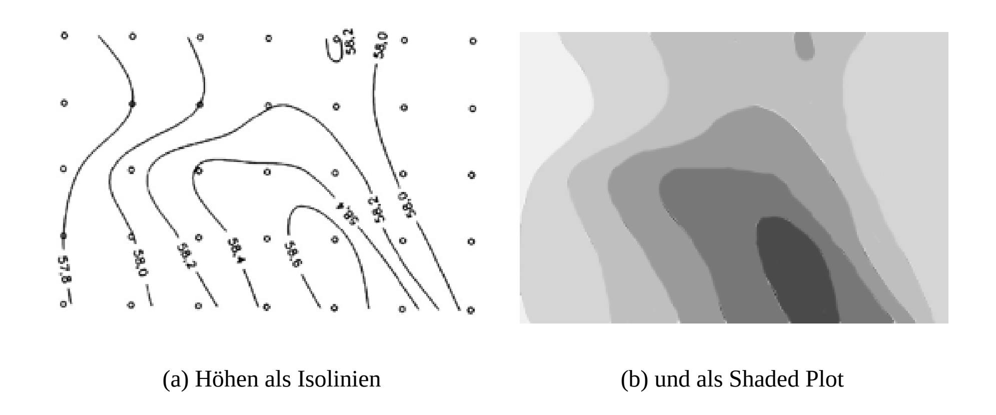
In diesem Fall muß man sich für einen dieser möglichen Verläufe entscheiden. Dazu interpoliert man den Funktionswert in der Mitte der Zelle, ist dieser kleiner als c, wählt man den 1. Fall, ist er größer als c, dann den 2. Fall. Es wird nach der Fläche zwischen den Isolinien entschieden. Die Schnittpunkte der Zellkanten werden durch lineare Interpolation zwischen den Eckpunkten bestimmt und miteinander durch ein Isoliniensegment verbunden. Danach geht man über zur nächsten Zelle in der Reihenfolge. Nachteile des Cell-Order-Verfahrens sind:

- Jeder Gitterpunkt und jeder ermittelte Stützpunkt der Isolinien wird doppelt besucht.
- Als Ausgabe erhält man lediglich Isoliniensegmente, welche noch nachbearbeitet werden müssen, um geschlossene Isolinien zu erhalten.
- Sich schneidende Isolinien können nicht erkannt werden.

Beim Contour-Tracing-Verfahren wählt man eine Zelle mit einem Vorzeichenwechsel, bestimmt das Segment, welches durch die Zelle verläuft, und wechselt dann in die benachbarte Zelle, in die die Isolinie hinein läuft. Die Isolinie verfolgt man so lange, bis entweder die Grenze des Gitters erreicht oder die Linie geschlossen wurde und wählt danach eine neue Zelle mit Vorzeichenwechsel, die man noch nicht besucht hat. Vorteil gegenüber dem Cell-Order-Verfahren ist, daß man bereits fertige Isolinien erhält. Bei verstreuten Daten wird normalerweise erst ein Rechteckgitter erzeugt, auf dem dann die oben genannten Verfahren angewendet werden.

Shaded Plots Shaded plots sind eine Weiterführung von Isolinien, wobei hier nicht die Linien, sondern die Flächen zwischen den Iosolinien durch unterschiedliche Farben/Schattierungen gezeichnet werden. Der große Vorteil dieser Technik im Gegensatz zur Isolinientechnik besteht

darin, daß unzusammenhängende Regionen mit gleichen Werten schneller erfaßt werden können, da sie die gleiche Farbe bzw. Schattierung besitzen. In Abbildung 2.19(a) kann man z. B. nicht sofort erkennen, daß der kleine Bereich in der Mitte der oberen Kante auf der gleichen Höhe wie ein anderer liegt. Aus Abbildung 2.19(b) hingegen ist dieser Umstand leicht ersichtlich. Als weitere Variante ist es auch möglich die Farbe der Flächen zwischen den Isolinien durch (lineare) Interpolation der Werte zu erhalten.

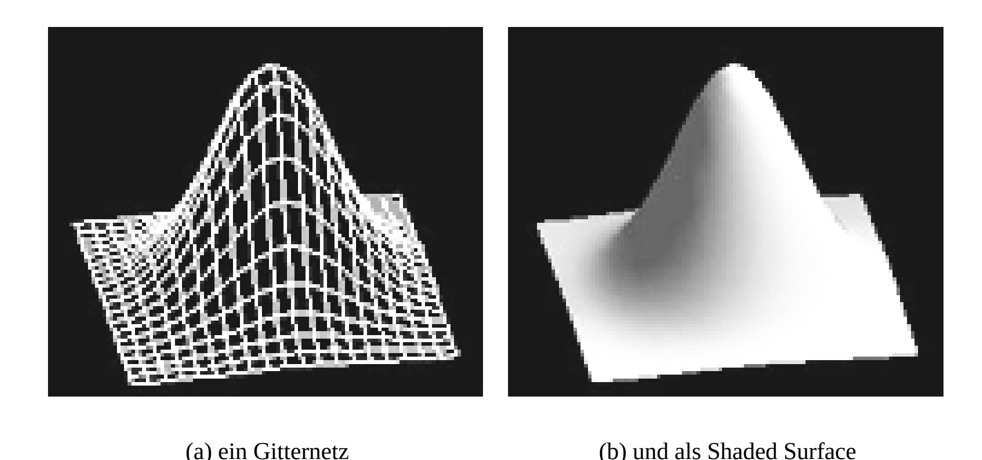


(a) Höhen als Isolinien (b) und als Shaded Plot

Abbildung 2.19: Isolinien und Shaded Plot

Gitternetze und Dreiecksnetze Eine Alternative zu den vorangegangenen Methoden ist es, Gitter- (engl. *rectangular*) bzw. Dreiecksnetze (engl. *triangular meshes*) über die Daten, die als Höhe über einer Ebene interpretiert werden, zu legen und damit ein Oberflächennetz zu erzeugen. Gitternetze werden meistens dadurch erzeugt, daß man in gewählten Abständen auf den Koordinatenachsen Gitterlinien parallel zur jeweils anderen Achse zieht. Dieses Verfahren eignet sich eigentlich nur für sowieso schon auf Rechteckgittern vorgegebene Samplewerte, da sonst entweder der Aufwand zur Interpolation der Samplewerte oder aber die Anzahl der Gitterlinien extrem zunimmt. Für Dreiecksnetze sieht das Verfahren etwas komplizierter aus, dafür hat man den Vorteil, daß sie für beliebig verstreute Daten effizient erstellt werden können. Denn aus jeder beliebigen Punktmenge läßt sich in $O(n \log n)$-Zeit (im ebenen Fall) ein Dreiecksnetz erstellen. Eine wohl bekannte Konstruktionsmethode zum Erstellen von Dreiecken aus beliebigen Punktmengen ist die von Delaunay (engl. *Delaunay triangulation*), nachzulesen u.a. in [1]. Dieses Oberflächennetz kann man dann auf eine Ebene projeziert oder über ein 3D-System ausgeben. Zusätzlich dazu werden nicht sichtbare Linien meistens aus der Ausgabe entfernt, was aber nicht unbedingt nötig ist. Vertiefende Literatur zu Netzen findet sich in [4].

Shaded Surfaces Beleuchtete Dreiecksnetze oder *shaded surfaces* bauen auf den Oberflächennetzen auf, indem zunächst eine einfarbige Textur über die Oberfläche gelegt wird. Danach werden die Normalenvektoren an den Oberflächenpunkten bestimmt und zum Schluß eine Lichtquelle eingefügt, welche die Oberfläche beleuchtet. Alternativ kann die Oberflächentextur natürlich beliebig gefärbt sein, z.B. ein Shaded Plot der Funktion (siehe auch bei Höhenfeldern 2.4.3).



(a) ein Gitternetz (b) und als Shaded Surface

Abbildung 2.20: Gitternetze un- bzw. beleuchtet

Höhenfelder Zum Abschluß der Techniken für zweidimensionale Funktionen seien hier noch kurz die Höhenfelder (engl. *height fields*) erwähnt, mit denen es möglich ist, zwei skalare Funktionen in einem Bild darzustellen, da die Daten nicht nur planar auf Isolinien oder Farben abgebildet werden, sondern zusätzlich auch auf eine dritte Raumkoordinate.
Im ersten Schritt wird zunächst für die erste Funktion ein Oberflächennetz bestimmt. Im zweiten Schritt wird aus der zweiten Funktion ein Shaded Plot erstellt, welcher als Textur über das Oberflächennetz der ersten Funktion gelegt wird. Diese Kombination der beiden Funktionen ist dann das Höhenfeld, wobei natürlich für beide Schritte dieselbe Funktion benutzt werden kann.

Techniken für dreidimensionale Funktionen

In diesem Fall wird eine dreidimensionale Funktion $(x_1, x_2, x_3) \mapsto f(x_1, x_2, x_3)$ an Punkten vorgegeben, wobei diese häufig die Punkte eines uniformen dreidimensionalen Gitters sind.

Voxelmodelle Ein Voxel (*volume element*) ist im Gegensatz zu einem Pixel (*picture element*), das einen Punkt in einer Ebene darstellt, ein Volumenelement im dreidimensionalen Raum und kann auf zwei unterschiedliche Weisen definiert sein:

- als Würfel, der einen Wert über den gesamten Inhalt repräsentiert, oder
- als Eckpunkt dieses Würfels. Werte der Punkte innerhalb des Würfels werden dann durch Interpolation bestimmt.

Voxelmodelle sind im Prinzip nichts anderes als dreidimensionale Rasterbilder und eignen sich so sehr gut zur Darstellung von Objekten, deren Inhalt ebenfalls beschrieben sein soll. Ein Voxelmodell ist ein dreidimensionales Array, das jedem Voxel einen Wert zuweist und damit beliebige Schnitte durch Objekte ermöglicht, was bei reinen Oberflächenmodellen nicht möglich ist, da diese ja im „Inneren" keine Informationen besitzen. Für die Darstellung dieser Voxelmodelle existieren zwei unterschiedliche Ansätze. Erstens die direkten Projektionsverfahren, die

das Voxelmodell auf eine darzustellende Ebene reduzieren und zweitens die Verfahren, die zuerst die Oberflächen im Voxelmodell suchen, um diese dann unterschiedlich auszugeben.

Isoflächen Ähnlich zu Isolinien stellen Isoflächen (engl. *isosurfaces*) gleiche Werte als Oberfläche dar. Die Schwierigkeit dabei besteht darin, die darzustellende Oberfläche zu finden. Eine Methode besteht darin, Konturen aus Schnittebenen zu rekonstruieren. Dies ist ebenfalls in [1] ausführlich erläutert. Eine andere Methode sind Voxel-Schnittmethoden, bei der für jedes Voxel anhand seiner acht Eckpunkte bestimmt wird, ob die Oberfläche ihn schneidet, und wenn, wie sie es tut. Bekannte Verfahren sind zum Beispiel *dividing cubes, marching tetrahedra* und *marching cubes*. Am Beispiel der *marching cubes* wird die Funktionsweise etwas näher betrachtet.

Cutplanes, Cutsurfaces Schnittebenen und Schnittflächen (engl. *cutplanes, cutsurfaces*) sind ein Spezialfall der *shaded plots* (s. Kap. 2.4.3). Aus einem dreidimensionalen Datensatz wird eine Ebene herausgeschnitten, und dann die Werte auf dieser Schnittebene per *shaded plot*-Verfahren dargestellt. Alternativ kann auch eine komplette Oberfläche extrahiert und Werte auf dieser Oberfläche visualisiert werden.

Oberflächenrekonstruktion am Beispiel *marching cubes* Die *marching cubes* werden so genannt, weil bei dieser Methode Voxel für Voxel im Raum abgegangen wird. Ähnlich dem Erstellen von Isolinien, werden zunächst alle Eckpunkte der Voxel klassifiziert, je nachdem, ob sie größer oder kleiner als der gesuchte Schwellenwert sind. Dann kann anhand dieser Klassifikation der Eckpunkte des Voxels wieder bestimmt werden, ob die Oberfläche das Voxel schneidet. Wie diese Oberfläche im Voxel aussieht, wird anhand einer Tabelle entschieden. Dazu werden die 8 Bit der Eckpunkte als Index zu dieser Tabelle genutzt, die die Lage der Oberflächen und Normalenvektoren für alle Fälle enthält. Da die dabei auftretenden 256 Möglichkeiten für die Oberfläche in einem Voxel aber durch Rotation und Spiegelung auf 15 verschiedene reduziert werden können, muß nur unter diesen 15 der passende Fall im Zusammenhang, um Löcher in der Oberfläche zu vermeiden, ausgewählt werden. Ähnlich wie bei der Isolinienerstellung werden jetzt die Schnittpunkte der Oberfläche auf den Voxelkanten durch lineare Interpolation bestimmt und anschließend die Oberfläche anhand der Schnittpunkte eingepaßt. Ist das Voxel fertig bearbeitet, wird zum nächsten übergegangen, bis alle Voxel besucht wurden.

Volume Rendering - Raycasting Volume Rendering bzw. Raycasting stellt die andere Herangehensweise zu Darstellung von Voxelmodellen dar. Da das Modell direkt auf eine Ebene projiziert wird, hat es sogar einige Vorteile gegenüber den Oberflächenrekonstruktionsverfahren, da zum Beispiel durchscheinende Objekte nicht plötzlich solide werden oder extra behandelt werden müssen. Stattdessen werden zunächst jedem möglichen Voxeltyp über Klassifikationstabellen ein Farb- und ein Transparenzwert zugewiesen. Dann wird das Voxelmodell entlang eines Sichtstrahles auf eine Ebene projiziert, wobei die Farb- und Transparenzwerte der einzelnen Voxel zu einem Pixel in der Sichtebene kombiniert werden. Um die Werte der Voxel, die ein Sichtstrahl entlangwandert, zu kombinieren, gibt es verschiedene Möglichkeiten, welche zu unterschiedlichen Ergebnissen führen.

- Maximalwertverfahren – Der Wert des Pixels wird durch den Maximalwert bestimmt, auf den der Strahl beim Durchwandern des Volumens trifft. Diese Methode wird z.B. zum

Erstellen von Magnetresonanz-Angiogrammen (engl. *magnetic resonance angiogram*) in der Medizintechnik benutzt.

- Durchschnittsverfahren – Der Wert des Pixels wird durch den Durchschnitt aller Voxelwerte bestimmt, die der Sichtstrahl trifft. Dadurch wird im Allgemeinen eine Art Röntgenbild des Volumens erzeugt.

- Additionsverfahren – Beim Additionsverfahren werden einfach die Transparenzwerte der Voxel addiert, während die Farbwerte meist gemischt werden, um den Pixel zu bestimmen. Diese Methode wird angewendet, um durchscheinende Bereiche über undurchsichtigem Material sichtbar zu machen.

- Schwellenwertverfahren- - Der Wert des Pixels wird einfach durch den ersten Voxelwert bestimmt, der über einem festgelegten Schwellenwert liegt. Diese Methode liefert prinzipiell ähnliche Ergebnisse wie eine Oberflächenrekonstruktion.

Um die Voxel zu bestimmen, die ein Sichtstrahl auf seinem Weg durch das Volumen trifft, wird entlang des Strahls in einer bestimmten Schrittweite durch das Volumen gegangen. Die Schrittweite ist meistens äquidistant, muß es aber nicht sein, um z.B. unwichtige Bereiche schneller bzw. wichtige Bereiche langsamer zu durchschreiten und so eine zonenabhängige Genauigkeit zu erzeugen. An jeder so erreichten Stelle des Strahls wird dann zunächst das Voxel bestimmt, durch den der Strahl gerade geht, was z.B. über den DDA-Algorithmus (*digital differential analyser*, [13]) geschehen kann. Danach wird der Wert des zu behandelnden Voxels bestimmt, welcher bei über Eckpunkten definierten Voxeln normalerweise durch trilineare oder trikubische Interpolation errechnet oder aber einfach der Wert des nächstgelegenen Voxels übernommen wird. Bei Voxeln, deren Wert den gesamten Inhalt bestimmt, kann man ihn direkt auslesen.

2.4.4 Techniken für vektorielle Daten

Einführung und Problembeschreibung

Zur Visualisierung von Vektorfeldern gibt es mehrere Ansätze: Die erste vorgestellte Methode ist trivial: In ausgewählten Punkten werden Vektoren als Pfeile (engl. *glyphs*) eingezeichnet, damit ist die Richtung des Vektorfeldes in diesem Punkt dargestellt. Zur Codierung der Geschwindigkeit gibt es mehrere Ansätze:

- durch die Länge der Pfeile, normiert auf eine Referenzgeschwindigkeit,

- durch die Farbe der Pfeile und die Angabe einer Farbtabelle, oder

- durch die Pfeildicke.

Gerade die erste Variante kann schnell zu unübersichtlichen Darstellungen führen. Um sich schnell einen Überblick über das Vektorfeld zu verschaffen, sind Pfeilplots jedoch konkurrenzlos effizient. Die Implementierung erfordert lediglich einige grundlegende Kenntnisse aus der Linearen Algebra.
Die Partikelverfolgungs-Technik (engl. *particle tracing*) läßt sich sehr elementar veranschaulichen: Gegeben ist ein Fluß, der innerhalb eines bestimmten Gebietes ruhig vor sich hin fließt oder in der Umgebung von Hindernissen Turbulenzen entwickelt. In diesen Fluß werden kleine

Markierungsbojen eingebracht. Bei diesem Experiment interessiert die Position der Markierungsbojen nach einer gewissen Zeit.
Statt Markierungsbojen werden auch Farbpartikel, bunte Tinte oder ähnliche Substanzen verwendet. Mit Hilfe dieser Visualisierung des Flusses sollen beispielsweise folgende Fragen beantwortet werden:

- Wann und auf welcher Bahn verläßt die Boje den betrachteten Teil des Flusses?
- Wird die Geschwindigkeit der Boje irgendwann null?
- Bleibt die Boje in einem Strudel gefangen, d.h. wiederholt sich ein Teil des Pfades der Boje?

Weiterführende Literatur zum Thema findet sich u.a. in [22, 23, 31].

Problemdefinition

Gegeben ist ein Gebiet (engl. *domain*) $\Omega \subseteq \mathbb{R}^2$ bzw. $\Omega \subseteq \mathbb{R}^3$. Auf diesem Gebiet ist ein Vektorfeld (engl. *vector field*) oder auch Geschwindigkeitsfeld (engl. *velocity field*) definiert, d.h. eine Funktion $v : \Omega \rightarrow \mathbb{R}^2$ bzw. $v : \Omega \rightarrow \mathbb{R}^3$. Jedem Punkt innerhalb des Gebietes sind eine Richtung und eine Geschwindigkeit zugeordnet. Betrachtet wird das Zeitintervall $T := [t_0, t_n]$ mit einer Startposition (engl. *seed location*) $\mathbf{p}_0 \in \Omega$ für das interessante Partikel zum Zeitpunkt t_0. Gesucht ist eine Funktion $p : T \rightarrow \Omega$, die der Bedingung $p(t_0) = \mathbf{p}_0$ genügt und die Position des Partikels im Rechengebiet zu jedem Zeitpunkt $t \in T$ beschreibt.
In der diskreten Variante besteht Ω aus einer Gitterzerlegung des Rechengebiets, d.h. einer endlichen Menge von Knoten (engl. *nodes*), die Elemente bzw. Zellen (engl. *cells*) definieren (s. Kap. 2.7). Elemente dieser Form heißen in der Literatur auch *curvilinear*. Ein Spezialfall sind Gitter aus achsenparallelen Einheitsquadraten bzw. Einheitswürfeln, man bezeichnet diese auch als kartesische Gitter (engl. *cartesian grids*). Das Geschwindigkeitsfeld v ist nicht mehr im ganzen Gebiet definiert, sondern nur noch in den gegeben Punkten (hier spricht man von Knotenwerten) oder einheitlich für jede Zelle (dies bezeichnet man als Zellwerte). Im folgenden werden nur Knotenwerte betrachtet. T ist kein Zeitintervall mehr, sondern eine endliche Folge von Zeitpunkten $\{t_0, \dots t_n\}$.

Durch die Diskretisierung ist das Problem in einem Teilaspekt komplexer geworden: Die Geschwindigkeit steht nur noch in Knoten zur Verfügung, und muß bei Positionen innerhalb der Zellen interpoliert werden, beispielsweise durch gewichtete Mittelung der Eckwerte der Zelle.

Wichtig ist die Unterscheidung zwischen zwei Arten von Geschwindigkeitsfeldern: In stationären Flüssen (engl. *steady flows*) ist, wie eben definiert, v zeitunabhängig, d.h. der Fluß bleibt zu jedem Zeitpunkt gleich. Im Gegensatz dazu ist im instationären Fall (engl. *unsteady flow*) der Fluß sowohl orts- als auch zeitabhängig ($v : \Omega \times T \rightarrow \mathbb{R}^3$), ändert sich also im Verlauf der Simulation.
Die analytische Lösung des Partikelverfolgungs-Problems genügt der folgenden, gewöhnlichen Differentialgleichung (exakter, dem folgenden Anfangswertproblem):

$$\frac{dp(t)}{dt} = v(p(t)), \quad p(t_0) = \mathbf{p}_0,$$

bzw. im instationären Fall:

$$\frac{dp(t)}{dt} = v(p(t), t), \quad p(t_0) = \mathbf{p}_0.$$

Physikalisch entspricht die erste Ableitung einer Ortsfunktion nach der Zeit einer Geschwindigkeit, dies erklärt die linke Seite der Gleichung. Auf der rechten Seite steht direkt die Auswertung des Geschwindigkeitsfeldes für denselben Zeitpunkt an derselben Position. Setzt man dies gleich, ergibt sich das obige Anfangswertproblem.

Der Basisalgorithmus

Im ersten Schritt muß die Zelle gefunden werden, in der sich die Startposition des Teilchens befindet (engl. *point location, cell search*). Solange sich das Teilchen im Rechengebiet befindet, wird zunächst die Geschwindigkeit des Teilchens an der aktuellen Position bestimmt (engl. *interpolation*), da die Geschwindigkeitswerte nur an den Eck-Knoten der Zelle bekannt sind, dann mit Hilfe obiger Gleichung die neue Position des Teilchens berechnet (engl. *integration*) und abschließend die neue Zelle bestimmt, in der das Teilchen sich nun befindet.

Darstellung

Generell gilt: Der Basisalgorithmus wird gleichzeitig auf eine gewisse Anzahl an Partikeln angewendet, und die verschiedenen, im folgenden erläuterten Visualisierungstechniken unterscheiden sich nur darin, wie die resultierenden Informationen kombiniert werden. Zur besseren Vergleichbarkeit werden die Techniken anhand desselben Simulationsdatensatzes illustriert, um direkt die Unterschiede in den berechneten Bildern ohne Ablenkung durch unterschiedliche Geometrien zu verdeutlichen. Dabei handelt es sich um einen oszillierenden Flugzeugflügel in 2D. Alle vier Bilder sind [23] entnommen.
Für stationäre Flüsse gibt es im wesentlichen eine Visualisierungtechnik: Für jedes Partikel separat werden die Tangenten an das Geschwindigkeitsfeld, die durch die Integration der Teilchen von einer Position zur nächsten approximiert werden, als Liniensegmente gezeichnet. In der Literatur findet sich dies als *streamlines*. Bei instationären Flüssen wurden drei Techniken



Abbildung 2.21: Streamlines

vorgeschlagen: *Pathlines* sind Trajektorien eines Teilchens, d.h. direkte Plots der Lösungsfunktion für den gegebenen festen Anfangswert. *Streaklines* sind Linien, die zu jedem Zeitpunkt alle aus derselben Startposition emitierten Partikel verbinden, mit anderen Worten, zu jedem



Abbildung 2.22: Pathlines

Zeitschritt wird aus jeder Startposition ein neues Teilchen emitiert und mit dem Vorgängerteilchen durch eine Linie verbunden. *Timelines* sind im Gegensatz dazu Linien, die Partikel



Abbildung 2.23: Streaklines

aus verschiedenen Startpositionen verbinden, die zum gleichen Zeitpunkt losgeschickt worden sind.



Abbildung 2.24: Timelines

Zellsuche

Bei der Zellsuche muß zwischen zwei verschiedenen Suchtechniken, welche von den gegebenen Informationen über die Partikel abhängen, unterschieden werden.

Wird ein Partikel in das Gebiet gesetzt, so sind außer der Position keine weiteren Informationen über das Partikel bekannt. In diesem Fall wird das Partikel mittels einer linearen Suche über alle Zellen gesucht. Somit wird für jede Zelle überprüft, ob sich das Partikel innerhalb dieser Zelle befindet. Diese Suchtechnik wird am Anfang einer Berechnung benutzt.
Wurde bereits ein Zeitschritt berechnet, ist zusätzlich zur aktuellen Position eines Partikels, seine vorherige Position bekannt und somit auch die Zelle, in welcher sich das Partikel im vorherigen Zeitschritt befand. Mit dieser Information kann nun eine Breitensuche gestartet werden, um die Position des Partikels zu ermitteln. Ausgehend von der Überlegung, daß während der Dauer eines Zeitschrittes das Partikel eine relative kleine Entfernung zurücklegt, wird nun über die Nachbarschaftsbeziehungen der Zellen, auf die angrenzenden Zellen zugegriffen und überprüft, ob das Partikel sich in eine von diesen Zellen befindet. Vereinfacht ausgedrückt, wird um die Zellen herum gesucht, bis das Partikel gefunden wird. Der Suchweg beschreibt in etwa eine Spirale.
Die oben genannten Techniken werden auf dem Grobgitter des Gebietes benutzt. Ist eine Zelle des Grobgitters gefunden, welche das Partikel enthält, wird im Feingitter (s. Kap. 2.2.5) weitergesucht.
Für jede Grobgitterzelle existiert ein Feingitter, welches nur Zellen enthält, die innerhalb dieser Grobgitterzelle liegen. Für die Suche eines Partikels innerhalb der Zellen des Feingitters, wird eine lineare Suche angewandt. Die Anzahl der Feingitterzellen innerhalb einer Grobgitterzelle ist, im Gegensatz zur Gesamtanzahl aller Grobgitterzellen in dem Gebiet, relativ klein, so daß eine lineare Suche eine tolerierbare Suchzeit erfordert. Durch die Verwaltung zusätzlicher Suchdatenstrukturen läßt sich Effizienz selbstverständlich noch steigern.

Geschwindigkeitsinterpolation

Zur Interpolation der Geschwindigkeitswerte im Inneren einer Zelle kann eine gewichtete Mittelung vorgenommen werden, beispielsweise mit dem Inversen des Abstandes zwischen Suchpunkt und Knoten (engl. *inverse distance weighting*), hier prototypisch für Hexaeder.
Falls in den acht Randknoten $\mathbf{n}_1, \ldots, \mathbf{n}_8$ der Zelle die Geschwindigkeiten $\mathbf{v}_1, \ldots, \mathbf{v}_8$ vorliegen, so ergibt sich die interpolierte, gesuchte Geschwindigkeit $v(\mathbf{x})$ im Punkt $\mathbf{x}$ im Inneren der Zelle als die gewichtete Summe

$$v(\mathbf{x}) = w_1\mathbf{v}_1 + \ldots + w_8\mathbf{v}_8,$$

wobei sich die Gewichte w_i in Abhängigkeit des euklidischen Abstandes $d_i = ||\mathbf{n_i} - \mathbf{x}||$ ergeben als

$$w_i = \frac{\frac{1}{d_i^2}}{\sum\limits_{j=1}^{8} \frac{1}{d_j^2}} \qquad \text{für } i = 1, \ldots, 8,$$

also als Verhältnis der Inversen des aktuellen Abstandes zum Gesamtabstand.
Für den Fall achsenparalleler Gitter kann trilineare Interpolation angewendet werden.

Lösung der DGL

In der numerischen Mathematik wurden viele Verfahren entwickelt, um gewöhnliche Differentialgleichungen mit Computern zu lösen. Für interessierte Leser sei auf das Vorlesungs-

skript [29] bzw. auf die Vorlesung [39] verwiesen, empfehlenswert sind auch die deutschsprachigen Lehrbücher [17, 18, 11].
Ein Ansatz besteht darin, den Ableitungsoperator durch eine Reihe von hinreichend kleinen Differenzenquotienten zu ersetzen. Andere Ansätze zerlegen einen Makroschritt in viele Teilschritte, und hangeln sich über diese Kleinschritte zur Lösung. Wieder andere verwenden primitive Startverfahren und intelligente Extrapolationsverfahren zur Verbesserung der schlechten Startwerte. Allen Verfahren gemeinsam ist jedoch die Tatsache, daß in der Zeit diskretisiert wird. Eine Klassifikation teilt gängige Verfahren dahingehend ein, ob sie neue Werte immer nur aus dem zuletzt berechneten, oder aus vielen vorherigen Werten errechnen. Man spricht hier von Einschritt- und Mehrschrittverfahren. Weiter unterscheidet man Verfahren mit konstanter Schrittweite, und Verfahren, die die Schrittweite adaptiv unter Verwendung intelligenter Fehlerschätzer anpassen, um den Fehler kontrollieren zu können. Als letzter Punkt sollte erwähnt werden, daß durch die Diskretisierung und die Verwendung von Computern viele neue Probleme im Bereich der numerischen Stabilität auftreten: Im wesentlichen können über lange Zeiträume verstärkende Rundungsfehler exponentielles Wachstum der Lösung verursachen, oder Verfahren konvergieren auf dem Computer nicht zur Lösung, obwohl dies analytisch beweisbar passieren sollte. Eine ausführliche Untersuchung existierender Verfahren besonders im Bezug auf Partikelverfolgung findet sich in [10]. Als besonders geeignet haben sich Verfahren des *Runge-Kutta*-Typs erwiesen, so zum Beispiel das folgende vierter Ordnung zur Schrittweite h:

$$\begin{aligned}
p^{(0)} &:= \mathbf{p}_0 \\
k_1 &:= h \cdot v(\mathbf{p}^{(k)}, t_k) \\
k_2 &:= h \cdot v(\mathbf{p}^{(k)} + \frac{1}{2}k_1, t_k + \frac{h}{2}) \\
k_3 &:= h \cdot v(\mathbf{p}^{(k)} + \frac{1}{2}k_2, t_k + \frac{h}{2}) \\
k_4 &:= v(\mathbf{p}^{(k)} + k_3, t_k + h) \\
\mathbf{p}^{(k+1)} &:= \mathbf{p}^{(k)} + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k4).
\end{aligned}$$

Dieses Verfahren konvergiert für $h \to 0$ wie $O(h^4)$ gegen die Lösung, d.h. man gewinnt pro Schritt mehrere Stellen Genauigkeit.

2.5 Verwendete Datenformate

2.5.1 Bilddateiformate

Im Bereich der Bildverarbeitung finden heutzutage ein Vielzahl von Bilddateiformaten Anwendung. Diese geben dem jeweiligen Anwender die Möglichkeit, abhängig von seinen weiteren Plänen, das für ihn sinnvollste Format zu verwenden. Er kann also ein Bildformat danach wählen, ob Plattenspeicher gespart werden soll oder ob eine bestimmte Detailtiefe des Bildes für die jeweilige Anwendung vorliegen muß. Jedes Pixel (Bildpunkt) einer 24-Bit Grafik benötigt 3 Byte Speicher. Manche Bildformate wie TIFF oder BMP behandeln Bilder 1:1 – d.h., daß sie ein Bild in der Größe speichern, die dem theoretisch errechneten Wert entspricht. Andere Formate wie TIFF-komprimiert, PICT oder Photoshop komprimieren Bilder ohne Informationsverlust. Dazu werden gleichfarbige Bildanteile zusammengefaßt. Dieses Verfahren spart

Plattenplatz und Übertragungszeit, verlangsamt aber das Öffnen der Datei durch ein Programm. Weitere Formate wie beispielsweise das JPEG-Format komprimieren Bildinformationen verlustbehaftet, d.h. „vergessen" Informationen, zugunsten höchster Verdichtungsraten.
Eine umfassende Übersicht über die hier nur zitierten Techniken bietet [15].

JPEG der Joint Photographic Experts Group

JPEG steht für „Joint Photographic Experts Group", die Gruppe, die JPEG ursprünglich spezifiziert hat. Dieses Datenformat komprimiert die Bildinformationen (von insbesondere Fotographien) derart, daß Komprimierungsraten von 10:1 bis 20:1 ohne sichtbaren Verlust der Qualität eines Bildes möglich sind. Bei weiterer Erhöhung der Komprimierungsrate, welche bei JPEG variabel ist, können Komprimierungsraten von 30:1 bis 50:1 mit nur geringen sichtbaren Qualitätseinbußen erreicht werden. Für Bilder, die weniger hohen Qualitätsansprüchen genügen müssen, kann auch noch mit einer Komprimierung von 100:1 gearbeitet werden. Bei der JPEG-Komprimierung werden die Bildinformationen zunächst in ein Format konvertiert, daß die Daten in Helligkeits- und Sättigungsgraden erfaßt (YCbCr, YUV, etc.), um diese Kanäle unterschiedlich stark zu komprimieren, da das menschliche Auge intensive Farbsättigung nicht so gut erfassen kann wie intensive Helligkeit. Die Auflösung der Farbsättigung wird dann je nach Komprimierungsgrad halbiert oder geviertelt. Anschließend wird mit Teilen der Bildinformationen (je 8x8 Pixel) eine Fouriertransformation durchgeführt, um ein Frequenzbild dieses Bildausschnittes zu erhalten. Von diesen Frequenzen werden dann (je nach Grad der Komprimierung) entsprechend viele (angefangen bei den höchsten) nicht mitgespeichert. Zu guter letzt durchlaufen die so verarbeiteten Daten noch eine Huffman-Komprimierung, um die Dateigröße noch weiter zu verkleinern. Zu beachten ist beim JPEG-Format, daß in jedem Falle Bildinformationen durch die Komprimierung unwiderruflich verlorengehen (*lossy compression*).

2.5.2 Filmdateiformate

QuickTime von Apple

QuickTime ist ein ungenau verwendeter Sammelbegriff für das QuickTime-Dateiformat, eine Menge an Applikationen und plug-ins, die dieses Format lesen und schreiben können, sowie eine komplette API. Das Format ist von Apple offengelegt worden, so daß es mittlerweile Software für die gängigsten Betriebssysteme, teils kommerziell, teils als open source gibt. Interessant – gerade für die Projektgruppe – ist die Integration der QuickTime API in die Java API, so daß mit nur wenigen Befehlen aus beliebigen Java-Programmen heraus Filme im QuickTime-Format generiert werden können. Teile der QuickTime-Spezifikation wurden mittlerweile von der ISO in den MPEG 4-Standard aufgenommen, dies kann als Zukunftssicherheit des Formats gewertet werden.
Ein QuickTime Film ist eigentlich ein *„wrapper"*, der lediglich angibt, welche Medien wann präsentiert werden sollen. Ein QuickTime Movie besteht aus einer oder mehreren Spuren (engl. *tracks*): Video-, Ton-, Text- oder andere Spuren. Spuren können synchronisiert werden, in Abfolge erscheinen, überlappen, sie können auch nach bestimmten Vorgaben alternieren (*alternates*). Die Vorgaben können unter anderem Sprache, Verbindungsgeschwindigkeit, Rechnergeschwindigkeit oder Benutzerverhalten sein. Ein QuickTime-Movie wird durch Addieren und Arrangieren von Spuren erstellt. Die in einem QuickTime-Film enthaltenen Medien können

in der Datei selber enthalten sein (*eigenständiger Film*), oder in ein oder mehreren abhängigen Dateien (*abhängiger Film*). Diese film-externen, abhängigen Dateien können sich auf dem selben oder einem anderen Computer befinden, der über das Internet erreichbar ist.
QuickTime-Spuren können bearbeitet werden, ohne abhängige Dateien zu verändern. Beim Sichern kann ausgewählt werden, ob ein Film abhängig oder unabhängig abgespeichert wird.
QuickTime kann Daten auf viele Arten komprimieren und dekomprimieren und versteht fast alle gängigen Formate. Diese Funktion kann sogar um neue Formate erweitert werden.
Zusätzlich unterstützt das QuickTime-Format noch *streaming*, d.h. Audio- oder Videodaten können bereits während der Übertragung abgespielt werden, im Idealfall bei hinreichend schneller Netzanbindung in Echtzeit. Beim *streaming* werden Daten auf dem Rechner des Benutzers nur gepuffert und nicht permanent gespeichert. Der Anbieter muß sich jedoch schon bei der Sicherung der QuickTime-Datei entscheiden, ob sie *streaming* unterstützen soll oder nicht.
Es gibt viel Literatur zu QuickTime, hier sei als Einstieg die Entwicklerseite von Apple selbst [28] empfohlen.

2.5.3 Gitter- und Geometriebeschreibungen

Geometriebeschreibungen können sehr große Unterschiede aufweisen:

Constructive solid geometry beschreibt Körper hierarchisch aus Grundkörpern zusammengesetzt, unter Verwendung von regularisierten Mengenoperationen.

Oberflächennetze beschreiben Körper durch ihre Oberfläche, entweder als Bezier- oder Splineflächen oder als Dreiecksnetze.

Implizite Funktionen stellen Körper als Nullstellenmenge einer oder mehrerer Funktionen dar.

Bei Gitterbeschreibungen ist nach den Fähigkeiten der zugrundeliegenden Numerikpakete zu unterscheiden: Dreiecksbasierte Gitter sind wesentlich einfacher zu generieren, Vierecksgitter zeichnen sich durch verbesserte numerische Stabilität aus. Für die Analoga im Dreidimensionalen gilt ähnliches. Bei beiden Richtungen werden Gitter jedoch immer als Knoten- und Elementlisten vorgegeben, die Formate unterscheiden sich dann noch darin, wie viele Zusatzinformationen, beispielsweise zu Nachbarschaftsbeziehungen, in die Gitterbeschreibung integriert wird.
Das von der Projektgruppe verwendete Format sollte zwei Kriterien erfüllen: offen und unter der GPL (s. [14]) frei verfügbar. Da diese Kriterien von proprietären Formaten nicht erreicht werden, wurde im Rahmen der Projektgruppe das bereits in FEAST im Einsatz befindliche Format FEAST gewählt bzw. diese Grundlage um eigene Anforderungen ergänzt. Es ist ein Maximaldatenformat aus geometrischer Sicht, d.h. es werden zu jeder Kante, jeder Fläche und jedem Element alle Nachbarschaftsbeziehungen mit in die Datei geschrieben. Dies macht das Format selbst für das Modul VISION interessant (s. Kap. 3.6). Die ausführliche Spezifikation findet sich im Anhang (s. Kap. D.1).

2.5.4 Formate für Simulationsergebnisse

Für die Übermittlung und Archivierung von Simulationsergebnissen gilt prinzipiell dasselbe wie für Gitterformate: Es gibt unüberschaubar viele. In der Projektgruppe wurde das GMV-Format (s. Kap. D.2) gewählt, weil es offen, frei und bereits an FEATFLOW angebunden ist.

2.6 Verwendete Programme und Hilfsmittel

2.6.1 Java und Java-Erweiterungen

Java 1.4.x

Die in der Projektgruppe größtenteils verwendete Programmiersprache ist Java (s. [41]) in der Version $1.4.x$. Vorteile von Java sind:

- freie Verfügbarkeit,
- Objektorientierung,
- Verfügbarkeit zahlreicher Erweiterungspakete, bspw. zur digitalen Bildsignalverarbeitung, und
- Netzwerkorientierung.

Früher hatte Java einen prinzipiellen Nachteil: Java-Compiler erzeugten keine Binärdateien, sondern lediglich ein maschinenunabhängiges Format, das zur eigentlichen Ausführung noch interpretiert werden mußte. Dies resultierte in Performanzeinbußen und einem relativ hohen Speicherbedarf zur Laufzeit. Moderne Java-Compiler kompilieren mittlerweile just-in-time, d.h. die performanzhemmende Interpretation findet nicht mehr statt.

Java Swing

Java Swing ist ein im Java-Sprachumfang enthaltenes Paket zur einfachen Erstellung von sehr flexiblen graphischen Benutzeroberflächen und damit die Weiterentwicklung des Java Abstract Window Toolkit (AWT). Da das AWT aus Geschwindigkeitsgründen auf Funktionen des jeweiligen Betriebssystems aufsetzte, war die Auswahl der Komponenten sehr beschränkt, denn nicht alle GUI-Bestandteile wurden auch von allen Systemen unterstützt.
Mit Java Swing wurde dieser Nachteil behoben. Es gibt nun die Möglichkeit, zwischen einem standardisiertem Java-Stil oder dem jeweiligen Betriebssystem-Stil zu wählen, und es stehen komplexere Datenstrukturen, wie Bäume und Tabellen – unabhängig vom Betriebssystem – zur Verfügung. Java Swing ist somit die Obermenge typischer plattformabhängiger GUI-Elemente, und nicht deren Schnitt.

Swing-Bestandteile Java Swing stellt eine Vielzahl an Komponenten zur GUI-Gestaltung bereit:

- Top-Level-Container: Applets, Fenster und Dialoge,
- Allgemeine Container: Panels, scrollbare Panels, trennbare Panels, Karteikartenreiter, Symbolleisten,
- Spezielle Container: Innere Fenster, überlagerte Fenster,
- Kontrollelemente: Schaltflächen, Auswahllisten, Listen, Menüs, Textfelder, Schieberegler, Hinweistexte, Beschriftungen, Fortschrittsbalken, Auswahldialoge, Tabellen, Textflächen, Baumansichten

- u.v.a.m.

Zusätzlich sind Ereignisroutinen für alle Ereignisse, die ein Benutzer interaktiv mit der Maus oder der Tastatur auslösen kann, vordefiniert und können einfach den Komponenten zugeordnet werden, z.B. Klickereignisse für Schaltflächen und Menüs, Drag&Drop-Ereignisse für Fenster etc.

Die Anatomie eines Java Swing-Programms Um eine GUI mit Java Swing zu erstellen, sind vier Schritte durchzuführen:

1. Die einzelnen Komponenten (Buttons, Listen, usw.) erzeugen und konfigurieren.
2. Die Komponenten in unterschiedliche Container einfügen (z.B. Fenster und Dialogboxen).
3. Die Komponenten innerhalb des Containers anordnen.
4. Festlegen, welche Komponenten welche Ereignisse auslösen, und Routinen zur Abarbeitung der Ereignisse schreiben.

Falls trotz der Fülle an Komponenten gerade die passende nicht zur Hand ist, können neue Komponenten einfach durch die Vererbung erzeugt werden.
Ausführliche Informationen zu Swing und ein gut zu verstehendes Tutorial nebst Referenz findet sich unter [35]. Swing ist Bestandteil jedes Java Software Development Kit ab Version 2 und muß nicht separat installiert werden.

Java 3D

Die Java 3D API (s. [42]) ist eine Sammlung von Klassen zur schnellen Erstellung von 3D Objekten, Animationen, sowie Interaktionen zwischen diesen. Java 3D ist szenengraphbasiert (s. Kap. 2.1.2).
Die Klassen unterteilen sich in Core-Klassen und Utility-Klassen. Die Utility-Klassen bieten dem Anwender schon teilweise vorgefertigte Lösungen, um den Aufwand zur Erstellung von 3D-Szenen auf ein Minimum zu reduzieren.
Java 3D bietet:

- vorgefertigte Klassen für viele Geometrieobjekte sowie Importfilter für gängige CAD-Formate,
- die Implementierung des Phong-Beleuchtungsmodells,
- fertige *behaviour nodes*, die einfach in den Szenengraph eingebunden werden können, um mausbasierte Navigation in der 3D-Szene zu ermöglichen,
- Exportfilter für gängige Bild- und Filmdateiformate (s. Kap. 2.5.1 und 2.5.2),
- voll konfigurierbare Projektionen auf ein Swing-Panel, das beliebig im eigenen Programm eingefügt werden kann,
- Hilfsklassen für geometrische Transformationen etc.

Java Media Framework API

Das *Java Media Framework API (JMF)* ist eine frei verfügbare Java-Erweiterung des Java-Herstellers SUN, die jedoch nicht zum Standard-Sprachumfang gehört. Das JMF kann unter `http://java.sun.com/products/java-media/jmf` kostenlos heruntergeladen werden.
Das JMF ergänzt den Java-Sprachumfang um Funktionen zur Audio- und Videobearbeitung. Zum Funktionsumfang zählen die Aufnahme, Wiedergabe, Rekodierung sowie das *streaming* von Mediadaten. Wie alle Java-Pakete ist das JMF plattformunabhängig.
Für die Projektgruppe wichtig ist die Unterstützung des QuickTime-Formats (s. Kap. 2.5.2) sowohl zur Kodierung, als auch zur Dekodierung.

Log4J

`Log4J` ist eine kostenlose Java-Klassensammlung zur vereinfachten Protokollierung eines Programmdurchlaufs. Anstelle konventioneller `System.out.println`-Aufrufe tritt hier die automatische Generierung von Logfiles. `Log4J` ist kostenlos unter
`http://jakarta.apache.org/log4j/docs/index.html`
erhältlich.

2.6.2 C und Fortran

GCC 3.2

Der Serverprozeß NEXUS wurde in der Programmiersprache ANSI C implementiert. Zur korrekten Übersetzung ist der GNU C-Compiler `GCC` in der Version 3.2 oder höher nötig. Er bietet auch sinnvolle Optimierungsoptionen sowie eine Überprüfung der Arrayzugriffe und steht für nahezu alle Architekturen und Betriebssysteme zur Verfügung.

Fortran 77, Fortran 90

FEATFLOW steht im Quelltext in Fortran 77 zur Verfügung, FEAST in Fortran 90. Zur Übersetzung sind Fortran-Compiler für diese Dialekte nötig, so beispielsweise die in der Gnu Compiler Suite `GCC` enthaltenen Programme `f77` und `f90`.

2.6.3 Entwicklungswerkzeuge

CVS 1.11.2

CVS, *concurrent versions system*, ist ein weitverbreitetes, frei verfügbares Programm zur Verwaltung eines Softwarepakets durch mehrere Entwickler (CVS kann alle textbasierten Dateiformate verwalten). CVS verwendet dazu eine zentrale Datenbank, das sogenannte *repository*. Quelltexte werden nie direkt in der Datenbank editiert, sondern immer auf einer lokalen Kopie. Fertige Dateien werden dann in die Datenbank zurücktransferiert (*commit*). Hierbei vergleicht CVS die neue Version mit der bereits in der Datenbank vorhandenen und versucht, die Änderungen automatisch einzupflegen. Haben aber beispielsweise zwei Entwickler an derselben Datei gearbeitet und konkurrierende Änderungen vorgenommen, so verweigert CVS das

Einchecken und der Entwickler muß manuelle Korrekturen vornehmen. So sichert CVS ein „sauberes” Repository.
Weitere Funktionalitäten sind die Verzweigung und Zusammenführung verschiedener Zweige der Entwicklung, so kann z.B. gefahrlos experimentiert und anschließend auf eine alte Version zurückgegriffen werden. Außerdem bietet CVS den Zugriff auf die Datenbank per SSH über das Internet an und so die Möglichkeit, weitverteilt an einem Projekt zu arbeiten. Die Standardreferenz zu CVS ist das Buch von Cederquist [7].

TogetherJ 6.0

Together [5] ist ein visuelles UML Modellierungswerkzeug für Softwareentwicklung in Java, C++ und weiteren Sprachen. Es arbeitet direkt mit den Quelltexten der erstellten Projekte und kann dadurch leicht mit existierenden Entwicklungsumgebungen und Versionskontrollsystemen kombiniert werden. UML steht für *unified modeling language* und bietet eine einheitliche Darstellungsweise für alle Phasen einer Softwarespezifikation: Fallstudien, Daten- und Kontrollflußdiagramme, Klassendiagramme, Sequenzdiagramme und viele andere mehr.
„The model is the code” ist der Grundgedanke von Together. Im Gegensatz zu anderen UML- und Entwicklungswerkzeugen werden die Modelle in Together direkt in die Quelltexte umgesetzt. Dies hat den Vorteil, daß die Änderungen an den Modellen direkt an den Quelltexten verfolgt werden können und umgekehrt. Zu den bedeutendsten Nachteilen zählt der hohe Verbrauch an Rechenressourcen von Together.
Together bietet die komfortable Erstellung von Sequenz-, „use case”-, Klassen- und vielen weiteren Diagrammen und generiert automatisch Coderahmen.

2.6.4 FEATFLOW und FEAST

FEATFLOW ist ein Programmpaket zur numerischen Strömungssimulation, konkret zur Berechnung der inkompressiblen Navier-Stokes-Gleichungen (s. Kap. 2.2) in 2D und 3D. FEATFLOW setzt auf die Basisbibliotheken `FEAT2D` und `FEAT3D` auf. Diese bieten effiziente Implementierungen für viele Unterprobleme, die typischerweise in der numerischen Strömungssimulation auftreten, wie z.B. Matrixaufbau, Löser etc. FEATFLOW ist komplett im Quelltext frei verfügbar unter `http://www.featflow.de` und wird ständig weiterentwickelt.
FEATFLOW ist im Vergleich zu kommerziellen Codes in einigen Bereichen bis zu einem Faktor 100 schneller in der Berechnung und kann durch seine Fortran77-Implementierung auf vielen Supercomputern eingesetzt werden. Es erfordert jedoch eine sehr lange Einarbeitungszeit, da ein Arbeitszyklus immer die Editierung vieler Parameterdateien und die Neukompilierung erfordert.
FEAST ist das Nachfolgepaket der Bibliotheken `FEATxD` und zeichnet sich durch eine inhärente Parallelisierung aus.
Alle Pakete wurden von S. Turek und seinem Team an den Universitäten Heidelberg und Dortmund seit Anfang der 90er Jahre entwickelt.

Kapitel 3

Systembeschreibung
Überblick und die Module im Detail

3.1 Allgemeine Beschreibung

Das von der Projektgruppe entworfene und implementierte System bietet eine Komplettlösung für den Bereich der numerischen Strömungssimulation. Bei der Spezifikation wurde besonders auf die Modularität, die Erweiterbarkeit und die Netzwerkfähigkeit geachtet. Der DEVISOR besteht aus den folgenden Komponenten:

CONTROL stellt die zentrale Konfigurations- und Steuerungseinheit des Systems dar. Alle anderen Module können über komfortable graphische Benutzerschnittstellen konfiguriert und gesteuert werden.

GRID ist ein Geometrie- und Gittereditor für zwei- und dreidimensionale Probleme.

NET bietet für jedes Modul eigene spezialisierte Netzwerkschnittstellen und sichert in Kombination mit dem Server NEXUS die Kommunikation zwischen verschiedenen Instanzen der Module über ein Computernetzwerk.

NEXUS ist der Server des gesamten Systems. Er bietet die Anbindung an alle Module und insbesondere die Anbindung an die verwendeten Numerikpakete FEATFLOW und FEAST.

VISION ist das Postprocessing-Werkzeug im DEVISOR-System und bietet verschiedene Visualisierungstechniken für Ergebnisdaten einer Simulation.

Die Serverkomponente NEXUS wurde in der Programmiersprache C implementiert, alle anderen Komponenten in Java. Die Auswahl der Programmiersprache ist durch die angestrebte Zielplattform begründet: NEXUS soll – wie auch die Numeriksoftware – auf Supercomputern lauffähig sein. Die übrigen Komponenten sollen überall plattformübergreifend einsetzbar sein, damit war Java die erste Wahl für die Implementierungssprache.

Die nun folgende allgemeine Beschreibung des Systems geschieht anhand einer prototypisch durchgeführten Simulation. Es wird bevorzugt Wert auf den Kontrollfluß zwischen den einzelnen Modulen gelegt, und nicht so sehr auf die einzelnen Methoden. Diese feine Detailstufe wird in den folgenden Unterkapiteln erläutert. Zuvor wird jedoch allgemein auf den Datenfluß zwischen den Modulen eingegangen.

3.1.1 Datenfluß zwischen den Modulen

Der Datenfluß von einem Modul zum anderen durchläuft folgende Stationen:

1. Als Reaktion auf die Betätigung eines GUI-Elements wird vom CONTROL-Modul aus eine Methode des **ControlWrappers** aufgerufen. Übergabeparameter sind Objekte aus dem **interim**-Paket.

2. Im Wrapper werden die übergebenen Informationen in einen Protokollbefehl übersetzt und an das Zielmodul versandt, konkret an die NEXUS-Instanz auf dem Zielrechner.

3. Je nach Befehl startet NEXUS auf dem Zielrechner ein neues Modul oder leitet den Befehl an den Wrapper des Moduls weiter.

4. Der Wrapper des Zielmoduls wandelt den Befehl wieder in eine objektbasierte Repräsentation um und übergibt die Daten an eine Methode des Zielmoduls.

5. Das Zielmodul reagiert entsprechend und generiert entweder eine Statusmeldung, eine Liste bereits generierter Ergebnisse oder ähnliches.

6. Diese Informationen gehen als Rückgabewert wieder an den Wrapper, der sie auf analoge Weise auf den Weg zum Kontrollmodul schickt.

Insbesondere gibt es für jedes Datum, das über das Netz verschickt werden soll, zwei äquivalente Repräsentationen: Programmintern werden die Daten objektbasiert bzw. record-basiert verwaltet. Für die Versendung über das Netzwerk gibt es jeweils eine Repräsentation als Teil des von der Projektgruppe spezifizierten Protokolls. Beide Darstellungen sind bijektiv ineinander überführbar, die Umwandlung übernehmen die Komponenten NEXUS und NET.

3.1.2 Beispielszenario

Es soll das Szenario aus Abbildung 3.1 betrachtet werden: Der Benutzer verwendet eine Instanz des CONTROL-Moduls auf seinem lokalen Computer. Er möchte eine Simulation auf einem entfernten Supercomputer durchführen, und die Ergebnisse direkt auf einer Graphikworkstation in Form einer Animation visualisieren lassen. Der gerenderte Film soll, ebenso wie die Zwischenergebnisse, die die Numeriksoftware produziert hat, auf seinen lokalen Rechner zur Weiterverwendung transferiert werden.
Eine genauere Detailstufe blendet die Netzwerkkomponenten ein: Auf jedem Rechner läuft eine Instanz des Servers NEXUS. Die in Java implementierten Module kapseln die Netzwerkfunktionalität jeweils durch eine Hüllklasse aus der NET-Komponente (*wrapper*). Das CONTROL-Modul und sein **ControlWrapper** tauschen beispielsweise Daten aus, indem das Modul Methoden des Wrappers aufruft, die jeweils eine vollständige Kommunikationssequenz (Senden der Anfrage und Empfangen der Antwort) durchführen und das Ergebnis in Form einer Klasse aus dem **interim**-Paket (s. Kap. 3.2.3) zurückliefern.

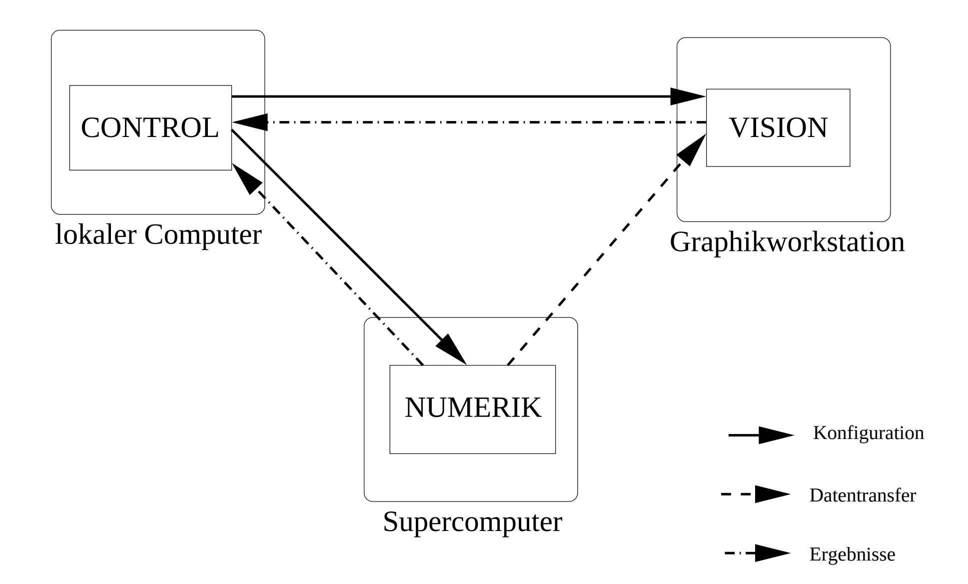


Abbildung 3.1: Beispiel-Szenario für eine mit dem DEVISOR durchgeführte Simulation

Phase 1 der Simulation: Vorbereitung Im folgenden wird davon ausgegangen, daß bereits Instanzen von NEXUS auf den verwendeten Computern gestartet wurden. Der Benutzer startet lokal eine Instanz des CONTROL-Moduls und trägt zunächst die Computer – identifiziert über Port und Namen bzw. IP-Adresse – in das Programm ein, mit denen er später kommunizieren möchte, in diesem Fall also den Supercomputer und die Graphikworkstation. Sodann wird vom Supercomputer eine Liste der angebotenen Module angefordert. Ein Numerikmodul wird ausgewählt und gestartet. Auf der Seite des Supercomputers wird ein Tochterprozeß von NEXUS initialisiert (*fork*), der von nun an exklusiv für die Kommunikation zwischen der CONTROL-Instanz und dem gestarteten Numerikmodul verantwortlich ist.

Der nächste Schritt besteht in der Konfiguration des Moduls. Sie geschieht in drei Schnitten: Zunächst wird eine Liste der unterstützten **Probleme** angefordert, der Anwender wählt beispielsweise das Membranproblem aus. Danach werden die nötigen **Parameter** eingestellt: Statistikoptionen, rechte Seite der Gleichungen usw. Ein spezieller Parameter ist das Grobgitter für die Rechnung. Das Numerikmodul liefert zwar ein Standardgitter als Vorbelegung für diesen Parameter, für eigene Rechnungen ist aber ein angepaßtes Gitter nötig. Also wird auf dem Rechner *localhost* eine Instanz des GRID-Moduls gestartet (mit denselben Mechanismen wie das Numerikmodul auf dem entfernten Rechner). Der Benutzer erstellt das individuelle Grobgitter, speichert es ab und trägt es als Wert des entsprechenden Parameters in die Konfiguration des Numerikmoduls ein.

Der letzte Vorbereitungsschritt besteht in der Einstellung der **problemunabhängigen Konfigurationsmöglichkeiten**. Im konkreten Beispiel des Membranproblems ist hier nichts zu tun.

Start der Berechnung Ist das Numerikmodul fertig konfiguriert, kann der Anwender die Berechnung starten. Das Modul hat bereits bei der Auflistung der angebotenen Probleme (s.o.) dem Kontrollmodul mitgeteilt, inwiefern es die Kassettenrekorder-Funktionen unterstützt, im Fall des Membranproblems werden alle Funktionen unterstützt. Der Anwender kann eine gestartete Rechnung nach Belieben pausieren und wieder aufnehmen. Falls dies in der Modulvorbe-

reitung so konfiguriert wurde, werden auf Knopfdruck Statistiken in Diagrammform angezeigt und es können eventuell anstehende Zwischenergebnisse heruntergeladen werden.

Darstellung von Zwischenergebnissen durch VISION Auf dem Rechner *localhost* wird eine Instanz des VISION-Moduls gestartet. Das Modul bietet nur ein Problem an, die Visualisierung. Seine Parameterliste gliedert sich in drei Bestandteile:

- Filter sieben ungewünschte Teile der Daten aus, beispielsweise mit der Rasenmähermethode jeden zweiten Datensatz oder alle Werte, die nicht auf einer bestimmten Isolinie liegen.
- Mapper überführen die Daten in eine Darstellungsform für den Bildschirm.
- Renderer stellen das fertige Bild dar.

Die globale Konfigurationsliste von VISION enthält die interessanteren Parameter: Zum einen wird das Grobgitter benötigt, auf dem auch das Numerikmodul arbeitet, zum anderen kann entschieden werden, ob nur ein Schnappschuß oder ein Film gerendert werden soll. Im konkreten Szenario wird sich der Anwender an dieser Stelle für einen Schnappschuß entscheiden. Der letzte Punkt ist die Einstellung der Datenquelle für VISION: Hier wählt der Anwender das Numerikmodul aus, das gerade im Hintergrund auf dem Supercomputer das Membranproblem löst. Ferner wird angegeben, welcher Zeitschritt visualisiert werden soll.
Im ersten Schritt fordert das VISION-Modul die zu visualisierenden Daten vom Numerikmodul an. Dies erledigt die angepaßte Hüllklasse aus dem Modul NET. Sind die Daten vorhanden, so werden sie in die interne Pipeline aus Filtern, Mappern und dem Renderer eingespeist und das Modul generiert den aktuellen Schnappschuß. Ist der Anwender mit dem Ergebnis – und der Analyse beispielsweise der Konvergenzrate anhand der Statistikdaten – zufrieden, wird er keine Änderungen vornehmen wollen. Andernfalls kann er einige der Parameter neu konfigurieren, die Berechnung um einige Zeitschritte zurücksetzen und mit den neu belegten Parametern korrigiert durchführen. Allerdings ist hierbei zu beachten, daß die Neubelegung der Parameter analog zu den Kassettenrekorderfunktionen explizit vom Modul unterstützt werden muß. Beim Membranproblem ist dies bei einigen Parametern der Fall.
Zu bemerken ist, daß es selbstverständlich möglich ist, die Verfolgung der Berechnung von beliebigen anderen Computern vorzunehmen. Auch ist es jederzeit möglich, die Verbindung zu laufenden Modulen zu trennen ohne diese pausieren oder gar beenden zu müssen. Die praktische Relevanz dieser Funktionalität ist offensichtlich.

Postprocessing Nach Ende der Berechnung beginnt das Postprocessing. Im betrachteten Fall wird der Anwender typischerweise eine Animation der gerechneten Zeitschritte vom VISION-Modul erstellen lassen. Die dazu notwendige Konfiguration geschieht analog, nur wird dieses Mal der *MovieMaker* des VISION-Moduls ausgewählt. Da Filmgenerierung eine aufwendige Aufgabe ist, wird der Benutzer das Modul dazu auf einer leistungsfähigen Workstation starten. Der Rohdatentransfer in Richtung VISION erfolgt wie oben beschrieben. Während die Animation erstellt wird, kann der Anwender die vom Numerikmodul generierten Ergebnisse und Statistiken auf seinen lokalen Computer transferieren.

Ende der Simulation Ist der Anwender sicher, die einzelnen Module und die von ihnen generierten Ergebnisse nicht mehr zu benötigen, so kann er einzelne Module oder das ganze Projekt (s.u.) beim Server löschen und somit nicht mehr benötigten Plattenplatz freigeben. Dies schließt einen Simulationszyklus ab.

3.1.3 Ergänzungen

In diesem Abschnitt werden einige zusätzliche Funktionalitäten des Systems erläutert.

Benutzerkonten Der DEVISOR erlaubt den Zugriff auf die einzelnen Daten nur in Abhängigkeit vom Benutzernamen. Ein Passwortschutz wurde allerdings noch nicht implementiert.

Projekte Der Benutzer kann seine verschiedenen Simulationen in verschiedene Projekte einordnen. Die oben beschriebene Membransimulation ist ein Beispiel für ein solches Projekt, macht aber bereits wichtigste Einschränkung dieses Zugangs deutlich: Daten können zwischen den verschiedenen Modulen nur ausgetauscht werden, wenn die Module demselben Projekt zugeordnet sind. Zusammen mit den Benutzernamen ist der DEVISOR somit klar ein System für den Einzelbenutzer und nicht für den Einsatz in Groupware-Umgebungen konzipiert. Es ist jedoch selbstverständlich möglich, mehrere Instanzen desselben Moduls unter einer Benutzerkennung innerhalb desselben Projekts parallel zu verwenden.

Modularität Das System hat zwei Kernkomponenten: Der Server NEXUS und das CONTROL-Modul sind für den Einsatz unerläßlich, und damit implizit auch Teile der NET-Komponente. Alle anderen Module sind austauschbar, insbesondere auch die Numeriksoftware. Ebenso selbstverständlich können eigene Module ergänzt werden, beispielsweise ein Gittergenerator als Ergänzung zum existierenden Gittereditor GRID. Im Anhang B.2 findet sich ein Tutorial zur Integration eines solchen neuen Moduls.

Programmierbarkeit Als Konsequenz des modularen Entwurfs sind auch die einzelnen Module wieder modular aufgebaut. Dies ermöglicht es, Teile des Systems in Form eines API auch in externen Programmen zu verwenden. Einige Beispiele sollen dies verdeutlichen:

- Darstellung eines Gitters oder einer Geometrie mit Hilfe von Komponenten aus den Modulen GRID oder VISION in beliebigen eigenen Fenstern.
- Einbindung einer Numeriksoftware durch hartcodierte Konfigurationen und die Verwendung des **ControlWrappers** isoliert vom CONTROL-Modul. Insbesondere können so Berechnungen ohne notwendige Benutzereingaben in eigene Programme eingebunden werden.
- Das Paket FOUNDATION stellt Klassen und Methoden zur Verfügung, um komplexe Gitter und Geometrien verwalten zu können. Hier könnte beispielsweise ein automatischer Gittergenerator aufsetzen.

3.1.4 Zustandsübergänge innerhalb der Module

Es gibt sechs verschiedene Zustände im Leben eines Moduls:

UNKNOWN für ungestartete Module,

STARTED für gestartete, aber unkonfigurierte bzw. noch nicht rechnende Module,

CONFIGURED für konfigurierte, noch nicht rechnende Module,

RUNNING für rechnende Module,

PAUSED für Module, die bereits gerechnet haben und aktuell pausiert sind, und

INACTIVE für bereits geschlossene Module, von denen allerdings noch Ergebnisse heruntergeladen werden können.

Die Regeln für die Zustandsübergänge sind wie folgt definiert (wobei die im Protokoll (s. Kap. D.3) spezifizierten Bezeichner die einzelnen Übergänge benennen):

STARTMODULE startet ein Modul,

CONFIGUREMODULE konfiguriert es,

START startet seine Berechnung,

PAUSE hält diese an,

STOP beendet sie,

FORWARD, REWIND, STEP spulen Berechnungen vor oder zurück,

CLOSEMODULE beendet ein Modul und

RESTARTMODULE erweckt es – falls das Modul dies unterstützt – wieder zum Leben.

Der Mealy-Automat in Abbildung 3.2 dokumentiert die konkreten Zustandsübergänge, der Operator $*$ soll dabei alle anderen, nicht explizit für den jeweiligen Zustand erwähnten Regeln ersetzen. Das Verhalten ist in diesen Fällen immer gleich: Es wird eine Fehlermeldung generiert.

3.2 Das Modul NET

Das Modul NET ist im Normalfall für den Anwender unsichtbar, da es über keine eigene Benutzeroberfläche verfügt. Es ist stattdessen eng an die einzelnen Module angebunden und stellt Methoden und Klassen zur Verfügung, um die Netzwerkkommunikation weitestgehend vor den Modulen zu verbergen. Dies ist insbesondere für Programmierer von Bedeutung, die eigene Systemerweiterungen implementieren und einbinden. Die Klasse **ControlWrapper** stellt gleichzeitig einen Großteil der DEVISOR-API zur Verfügung.

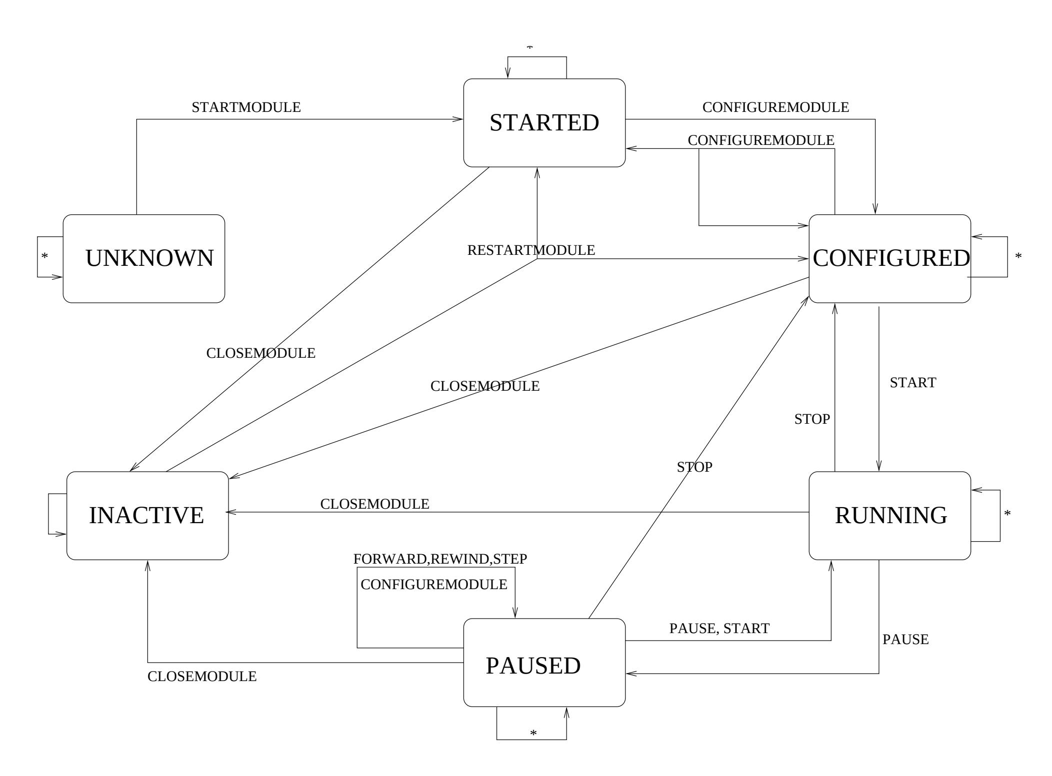


Abbildung 3.2: Zustandsübergangsdiagramm

3.2.1 Paketstruktur und Modulbeschreibung

Das Modul NET wurde getreu zweier Paradigmen entworfen:

- *Netzinterna bleiben den Modulen vollständig verborgen.* Der Vorteil dieses Paradigmas ist offensichtlich: Die Modularität und Erweiterbarkeit des Systems wird so auch auf die Netzwerkkommunikation ausgedehnt. Durch die Verwendung von abstrakten Klassen und einem ganzen Paket (s. Kap. 3.2.3) zur Protokollkapselung können Erweiterungen am Protokoll ohne großen Aufwand und insbesondere auch ohne Konsequenzen für bereits angebundene Module vorgenommen werden.

- *Alle Kommunikationssequenzen sind beantwortet.* Dies bedeutet, daß jede Kommunikation zwischen zwei Modulen immer von einem Modul ausgeht und definitiv vom anderen Modul beantwortet wird. Dies vereinfacht zum einen die Implementierung der Module erheblich und garantiert die Fehlerkontrolle über das Netzwerkmodul. Außerdem hält dies die Schnittstelle zu den einzelnen Modulen schmal und übersichtlich. Im folgenden wird eine solche Kommunikationssequenz auch synonym mit Protokollsequenz bezeichnet.

Das Modul NET besteht aus fünf Paketen:

devisor.net enthält die wichtige Schnittstellenklasse **NetException**, über die alle Fehlermeldungen des Moduls an die Außenwelt weitergegeben werden.

devisor.net.interim enthält Klassen zur Kapselung der Protokollinterna, insbesondere die Parser, die Klassen aus diesem Paket in ihre Protokollrepräsentation umwandeln und umgekehrt.

devisor.net.wrappers enthält die Hüllklassen um die Module.

devisor.net.example enthält eine komplette Beispielapplikation als Tutorial zur Integration neuer Module.

devisor.net.debug enthält Testroutinen für die skriptbasierte Steuerung aller Module sowie den **DeViSoRBenchmark**, die umfangreiche Testumgebung für alle Module.

3.2.2 Fehlermeldungen

Alle Fehlermeldungen – seien es Parserfehler, Verbindungsfehler oder sonstige – des NET-Pakets werden in einer Klasse gesammelt: die Klasse **NetException**. Sie erweitert die Klasse **Exception** aus der Java-Klassenbibliothek um folgende Funktionalitäten:

- Angabe der Klasse, in der der Fehler erstmals auftritt,
- Fehlerliste, um mehrere Fehler zu bündeln und
- Kaskadierbarkeit.

Alle Methoden des gesamten Pakets werfen – sofern sie Kontakt zum Netzwerk haben – eine Instanz dieser Exception. Insbesondere bei den verschachtelten Parsern (s.u.) ist diese Klasse sehr hilfreich: Auftretende Exceptions müssen lediglich an den Konstruktor dieser Klasse übergeben werden, um automatisch mitverwaltet und durch die Verschachtelungshierarchie zum Anwender gereicht zu werden.

3.2.3 Die Protokollkapselung durch das Paket interim

Das Paket **interim** enthält eine große Anzahl an Objekten zur Kapselung der Protokollinterna und insbesondere der Parser, die die beiden Datenrepräsentationen – objektbasiert und protokollbasiert – ineinander umwandeln. Bezüglich des implementierten Protokolls sei auf Anhang D.3 verwiesen.
Die Idee hinter diesem Paket ist einfach zu verstehen: Im wesentlichen gibt es zu jedem Teil einer Protokollsequenz eine eigene Klasse, die sich selbst aus einem Datenstrom rekonstruieren und auch zurück in das Protokollformat schreiben kann. Zu jeder Klasse gibt es also eine äquivalente Repräsentation in Form eines Auszugs aus einer Protokollsequenz. Konkret muß die jeweilige Klasse dazu das Interface **Parsable** aus diesem Paket implementieren, das die beiden Methoden **toProtocol** und **fromProtocol** vorschreibt.

```
public void toProtocol (PrintWriter outstream)
      throws devisor.net.NetException;

public void fromProtocol (BufferedReader instream)
      throws devisor.net.NetException;
```

Codebeispiel 1: Das Interface ***Parsable***

Übergabeparameter ist jeweils ein Datenstrom. Die Kontrolle über den Datenfluß liegt dabei allerdings immer bei den übergeordneten Klassen aus dem Paket **wrappers**. Ein Beispiel soll dies verdeutlichen, wobei die verwendeten Klassennamen erst in den folgenden Abschnitten erklärt werden: Im Protokoll gibt es einen Befehl, der beim Server die Liste der vom jeweiligen Modul benötigten Parameter anfordert. Das Parsen der Antwort läuft nach folgendem Schema ab: Der **ControlWrapper** (s.u.) instantiiert ein Objekt vom Typ **Header**, trägt die benötigten Parameter wie Projektkennzeichner und Benutzername ein und übergibt dessen **toProtocol**-Methode den Datenausgangsstrom aus der Socket, die an den Server gebunden ist. Die Instanz des **Headers** schreibt den äquivalenten Protokollblock in den Datenstrom. Danach sendet der Wrapper den Protokollbefehl `GetParameterListFor` (s. Kap. D.3). Das Senden an den Server endet mit der Instantiierung und Versendung eines Objekts vom Typ **Footer**. Damit ist der erste Teil der Kommunikationssequenz vollständig. Die Hüllklasse erzeugt danach eine leere Instanz der Klasse **Header** und übergibt den Dateneingangsstrom derselben Socket an die entsprechende **fromProtocol**-Methode. Diese blockiert, bis sie die erste Zeile des Header-Protokollblocks empfängt und liest den vollständigen Block aus. Der Kontrollfluß geht zurück an den **ControlWrapper**, sobald das spezielle *Stop-Wort* `End_Header` gelesen wurde. Danach können diese Informationen ausgewertet und beispielsweise eine Fehlermeldung generiert werden, wenn der übermittelte Projektkennzeichner nicht mit dem erwarteten Projektkennzeichner übereinstimmt. Danach wird ein leeres Objekt vom Typ **ParameterList** instantiiert und dessen Parser-Methode aufgerufen. Diese Methode iteriert über die vorgefundenen Abschnitte, die jeweils einen Parameter beschreiben, und ruft je nach gelesenem Typ die entsprechende Methode einer der Erweiterungen der **Parameter**-Klasse auf, so zum Beispiel **ScalarParameter.fromProtocol** oder **SelectionParameter.fromProtocol**. Treten im Datenstrom keine Parameterbeschreibungen mehr auf, so geht die Kontrolle zurück an den **ControlWrapper**, der die Protokollsequenz durch analogen Aufruf der **Footer.fromProtocol**-Methode beendet.

Dieser Kontrollfluß beinhaltet auch gleichzeitig eine einfache Fehlerkontrolle: Jede Klasse, die das Interface **Parsable** implementiert, „kennt” ihre äquivalente Repräsentation als Protokollsequenz, und sobald eine Protokollzeile auftritt, die im Zusammenhang der gerade erwarteten Protokollsequenz keinen Sinn ergibt, kann die entsprechende Interimsklasse eine Fehlermeldung generieren und an die aufrufende Klasse weiterleiten. Insbesondere sind die beiden Methoden des Interface **Wrappable** immer bijektiv.

Das Paket stellt Klassen für die folgenden Datencontainer zur Verfügung (für Protokollrepräsentationen und das konkrete API wird auf die Anhänge D.3 und C.5 verwiesen):

- Protokollheader und -Footer,
- Modulbeschreibungen, Modullisten und Modulstatusberichte,
- Problembeschreibungen, Parameterlisten und globale Konfigurationslisten,
- Parametertypen: skalare und vektorielle Werte, Gitter und Geometrien, m aus n - Auswahlmöglichkeiten, Funktionsdeklarationen und Gruppierungsparameter,
- GUI-Beschreibungen,
- Statistiken und Statistiklisten,
- Ergebnisse und Ergebnislisten,

- Fehler und Fehlerlisten, die auch als Statusmeldungen verwendet werden können, sowie
- Datenquellen-Beschreibungen.

Klassen, die Listen von verschiedenen Interimsobjekten kapseln, bieten zusätzlich die üblichen Listenoperationen wie *„hinzufügen", „löschen"* oder *„durchlaufen"*.
Alle Parameterklassen beinhalten ergänzend zwei weitere Methoden, weil das Protokoll vorschreibt, die in ihnen gekapselten Informationen auf zwei verschiedene Weisen über das Netzwerk zu versenden: Die normalen, durch das Interface **Wrappable** vorgeschriebenen Methoden parsen jeweils die vollständige Beschreibung, die **set**-Varianten nur eine Kurzbeschreibung, die einem Kennzeichner den vom Benutzer ausgewählten Wert zuweisen. Der Grund ist offensichtlich: Auf dem Weg vom Modul zum CONTROL-Modul wird beispielsweise für jeden Parameter seine GUI-Repräsentation übertragen. Der Wert, den der Benutzer einträgt, wird zurück an das Modul übertragen. Hier ist selbstverständlich keine vollständige GUI-Beschreibung mehr nötig, und es kann Bandbreite gespart werden. Die Methoden sind so implementiert, daß sie auf die gleichen Instanzen der Parameterklassen aufgerufen werden können und nur die geänderten Werte bzw. Benutzerselektionen eintragen. Weiter hat jeder Parameter eine Identifikationsnummer, die nicht global, aber lokal für jedes Modul eindeutig sein müssen. Für bestimmte Situationen sind auch feste Kennzeichner vorgeschrieben, Details sind der Protokollspezifikation (s. Kap. D.3) zu entnehmen. Ein Beispiel: Jedes Modul darf nur eine Instanz der Klasse **DataSourceParameter** verwenden, sie hat die feste Nummer 2. Diese Klasse konfiguriert die Datenquelle für das Modul, also beispielsweise für VISION, von welchem Numerikserver Rohdaten zur Visualisierung angefordert werden sollen.
Die Klasse **Module** stellt nicht nur eine Beschreibung der eigentlichen Modulfunktionalität dar, sondern beinhaltet zusätzlich Attribute zur Speicherung des Zielrechners, des Ports, der Socket, der assoziierten Datenströme und des assoziierten **KeepAliveThreads** (s.u.).
Damit die Interimsobjekte in eine funktionierende dynamisch erzeugte graphische Benutzeroberfläche transferiert werden können, sind die im Protokoll (s. Kap. D.3) spezifizierten Regeln zu beachten.

3.2.4 Die Schnittstelle zu CONTROL

Für das CONTROL-Modul bietet das NET-Paket eine spezielle Hüllklasse, den **ControlWrapper**. Für jeden Protokollbefehl stellt diese Klasse eine eigene Methode bereit. Jede dieser Methoden führt die komplette, geforderte Protokollsequenz aus und gibt das Ergebnis der Kommunikation im erfolgreichen Fall als Instanz einer der Klassen aus dem **interim**-Paket und sonst per **NetException** zurück. Insbesondere implementiert diese Klasse das vollständige Protokoll. Damit ist durch Kombination einiger der angebotenen Methoden auch die Programmierbarkeit gewährleistet.
Einer Instanz des CONTROL-Moduls ist immer ein **ControlWrapper** zugeordnet, der trotzdem die Kommunikation zu beliebig vielen Modulen bietet. Dies geschieht jeweils über die den Methoden übergebene Referenz auf ein **Module**-Objekt (s.u.). Über diese Referenzen ist der **ControlWrapper** auch mit Hilfe eines einfachen lock-Mechanismus threadsafe implementiert worden. Die Methoden, die eine neue Verbindung verwenden und diese wieder schließen, sind schon per Definition threadsafe. Alle anderen warten auf ein lock, und erst wenn die Verbindung frei ist, wird sie von der nächsten Methode reserviert.

Klassifizierung der Methoden

Es ist zwischen fünf verschiedenen Befehlsgruppen zu unterscheiden:

- Die Methoden **getAvailableModules**, **getRunningModules** und **getInactiveModules** dienen dazu, unterstützte Module, bereits gestartete Modulinstanzen oder beendete, aber noch nicht gelöschte Instanzen vom Server anzufordern. Sie erhalten als Übergabeparameter den Rechnernamen und den Port des Zielmoduls sowie Benutzername und Projektkennzeichner des Anwenders. Es wird eine temporäre Verbindung auf- und nach Ende der Kommunikationssequenz wieder abgebaut. Die per Parameter übergebenen Informationen werden in jede Instanz der Klasse **Module** eingetragen, die diese Methoden in Form einer **ModuleList** zurückliefert.

- Die Methoden **startModule** und **connectModule** starten eine Modulinstanz neu oder nehmen Verbindung zu einem bereits laufenden Modul auf. Ihr Übergabeparameter ist jeweils eine Instanz der Klasse **Module**, die von den Methoden **getAvailableModules** bzw. **getRunningModules** oder **getInactiveModules** zurückgeliefert wurden. Beide Methoden bauen eine neue Verbindung auf und starten den **KeepAliveThread**. Die verwendete Socket und ihre Datenströme sowie der Thread werden in die übergebene **Module**-Instanz eingetragen.

- Die Methoden **closeModule** und **disconnectModule** sind genau invers dazu und erhalten demzufolge eine **Module**-Instanz.

- Die Methode **downloadResult** baut, basierend auf der übergebenen **Module**-Instanz, die bereits eine bestehende Verbindung kapselt, eine neue parallele Verbindung auf, um den Ergebnistransfer nicht-blockierend zu realisieren. Diese temporäre Verbindung wird beim Beenden der Methode wieder abgebaut.

- Alle anderen Methoden erhalten als Übergabeparameter eine bereits verbundene **Module**-Instanz. Der darin laufende **KeepAliveThread** wird kontrolliert beendet, die entsprechende Kommunikationssequenz abgearbeitet und am Ende wird der **KeepAliveThread** wieder gestartet.

Struktur einer Methode

Da eines der Entwurfsparadigmen fordert, daß jede Protokollsequenz beantwortet sein soll, haben alle diese Methoden dieselbe Struktur:

1. Binden der Socket an das Zielmodul oder Extraktion der bestehenden Verbindung aus dem **Module**-Objekt.

2. Eventuell Beenden des laufenden **KeepAliveThreads**, sofern nötig.

3. Instantiierung eines **Header**-Objekts mit Projektkennzeichner und Benutzername und Aufruf seiner **toProtocol**-Methode.

4. Generierung und Versendung der Anfrage an das Zielmodul.

5. Instantiierung einer **Footer**-Instanz und Aufruf der **toProtocol**-Methode.

6. Warten auf die Antwort.

7. Einlesen des **Headers** der Antwort.

8. Instantiierung eines Interimsobjekts, das den relevanten Teil der Antwort kapselt, und Aufruf seiner **fromProtocol**-Methode.

9. Extraktion des **Footers**.

10. Weiterverarbeitung der Antwort und entweder Rückgabe des entsprechenden Interimsobjekts oder Werfen einer **NetException**.

Der KeepAliveThread

Da `TCP/IP`-Verbindungen nicht automatisch aufrecht gehalten werden, und besonders um auch asynchron zu den oben aufgelisteten Methoden Nachrichten vom Zielmodul zum CONTROL-Modul transferieren zu können, wurde der **KeepAliveThread** spezifiziert. Er wird am Ende der oben aufgelisteten Methoden automatisch gestartet und zu Beginn des nächsten Methodenaufrufs wieder beendet und läuft asynchron zu den Aufrufen der **ControlWrapper**-Methoden. In beliebig einstellbaren Zeitintervallen generiert dieser Thread eine spezielle Protokollsequenz und sendet sie über die im Konstruktor per **Module**-Referenz übergebene Verbindung an das Zielmodul.
Drei Szenarien werden als Antwort unterstützt:

1. Das Modul antwortet mit einer speziellen Protokollsequenz, dies entspricht dem klassischen *ping*-Signal.

2. Das Modul sendet angefallene Fehler- oder Statusmeldungen in Form einer **ListOfErrors** im Huckepack-Stil zurück.

3. Das Modul schickt anfallende Statistikdaten zurück, falls es diese Option als Parameter bietet.

In den letzten beiden Fällen werden die empfangenen Objekte in der *inbox* des **ControlWrappers** eingeordnet. Dies ist eine Instanz der Klasse **NotificationQueue**, die eine synchronisierte Warteschlange für das *single-consumer-multiple-producers*-Szenario implementiert und im wesentlichen die zwei Methoden für das Einhängen und die Extraktion von Objekten bietet. Aus Gründen der besseren Zuordnung zu den einzelnen Modulen (zur Erinnerung: ein **ControlWrapper** bietet die Verbindung zu beliebig vielen Modulen jeweils über die den Methoden übergebenen **Module**-Instanzen) werden zusätzlich noch der Projektkennzeichner und die Modulidentifikation eingetragen.

Programmierbarkeit und API

Durch Kombination einiger der vom **ControlWrapper** gebotenen Methoden lassen sich Teilfunktionalitäten der einzelnen Zielmodule in eigene Programme integrieren. Dies soll hier am Beispiel einer numerischen Berechnung verdeutlicht werden:

1. Start des Servers NEXUS mit integriertem Numerikmodul.

2. Holen der Modulliste mit **getAvailableModules**.
3. Extraktion des Numerikmoduls.
4. Start des Numerikmoduls per **startModule**.
5. Holen der Problem- und Parameterlisten durch **getProblemList**, **getParameterListFor**, **getConfigurationList**.
6. Hartcodierte Anpassung gemäß der eigenen Vorstellungen.
7. Konfiguration des Moduls per **configureModule**.
8. Start der Berechnung per **start**.
9. Beenden des Moduls nach Rechnungsende per **stop**.
10. Herunterladen der Ergebnisse per **getAvailableResult** und **downloadResult**.
11. Schließen des Moduls per **closeModule**..

Beispielimplementierungen hierzu finden sich im **debug**-Package.

3.2.5 Die Schnittstelle zu beliebigen anderen Modulen

Das Paket NET bietet fertig implementierte abstrakte Klassen zur Integration beliebiger anderer Module. Diese Klassen finden sich ebenfalls im Paket **wrappers**. Ein ausführliches Tutorial zur Modulintegration findet sich im Anhang B.2.
Die Klasse **AbstractModuleWrapper** wird per Konstruktor an einen freien Port, der beim Betriebssystem angefordert wird, gebunden. Dieser Port wird dann über eine spezielle Protokollsequenz an die aufrufende NEXUS-Instanz zurücktransferiert. Das aufrufende Modul muß das Interface **Wrappable** implementieren und ist verantwortlich, die benötigten Referenzen korrekt zu setzen. Sobald dies geschehen ist, kann die Initialisierung durch den Aufruf der beiden Methoden **updateLocalization** und **startCommunication** beendet werden. Die erste Methode sichert die Konsistenz der zugrundeliegenden Sprachdateien zwischen Modul und Hüllklasse, die zweite Methode startet eine Endlosschleife, in der die Hüllklasse neue Verbindungen akzeptiert und basierend auf diesen eine neue Instanz der Klasse **AbstractCommunicationThread** startet. Eine weitere wichtige Funktionalität, die in dieser Klasse gekapselt ist, ist die Bereitstellung einer *outbox*, einer Instanz der Klasse **NotificationQueue**, die oben erläutert wurde. In die *outbox* kann das Modul Statusmeldungen und Fehlerlisten (als Instanzen der Klasse **ListOfErrors**) einhängen oder Statistiken zur Versendung an das aufrufende CONTROL-Modul anmelden. Diese Objekte werden dann, wie oben beschrieben, im Huckepack-Stil beim nächsten *ping*-Signal übermittelt.
Die Klasse **AbstractCommunicationThread** implementiert das vollständige Protokoll: Es wird zunächst der Protokollheader eingelesen. Basierend auf der nächsten Zeile wird entschieden, welcher Protokollblock folgt, und in die entsprechende Routine verzweigt. Für jede Protokollsequenz steht eine dieser Hilfsmethoden zur Verfügung. Diese Methoden arbeiten alle nach demselben Schema:

1. Einlesen der Anfrage.

2. Generierung der Antwort bzw. Anfordern der Antwort beim assoziierten Modul über die vom Interface **Wrappable** vorgeschriebenen Methoden.
3. Versenden der Antwort.
4. Beenden der Verbindung zum Server und des Threads.

Die Kommunikation zwischen CONTROL-Modul, NEXUS und Zielmodul läuft folgendermaßen ab: Die Verbindung zwischen Server und CONTROL-Modul bleibt permanent bestehen. Zwischen Server und Zielmodul wird für jede Kommunikation eine neue Verbindung geöffnet. Auftretende Fehler werden in eine Instanz der Klasse **ListOfErrors** umgewandelt und automatisch in die *outbox* des Wrappers eingehängt.

3.2.6 Die Schnittstelle zu GRID

Basierend auf diesen abstrakten Klassen stellt das Paket NET eine angepaßte Erweiterung durch den **GridWrapper** bzw. **GridCommunicationThread** zur Verfügung. Diese Klassen leiten im wesentlichen nur nicht unterstützte Protokollsequenzen auf die spezielle Sequenz `UnsupportedCommand` um. Die Implementierung der Erweiterung ist unaufwendig und bedarf keiner weiteren Erläuterung. Das GRID-Modul implementiert dazu das Interface **Wrappable**.

3.2.7 Die Schnittstelle zu VISION

Die Schnittstelle zu VISION ist etwas umfangreicher als die zu GRID und wird analog durch die beiden Klassen **VisionWrapper** und **VisionCommunicationThread** realisiert. Letztere unterscheidet sich nur unwesentlich von ihrem GRID-Pendant. Die Klasse **VisionWrapper** stellt jedoch eine wichtige Methode zur Verfügung: **getTimeStepData**. Diese Methode bekommt eine Instanz von **DataSourceParameter** übergeben sowie den Zeitschritt, zu dem Rohdaten transferiert werden sollen. Mit den Informationen aus dem Parameter wird Kontakt zu dem dort bezeichneten Numerikserver aufgenommen und ein zeitschrittabhängiger Datensatz transferiert, der dann lokal zwischengespeichert wird, so daß das VISION-Modul ihn als Datenquelle verwenden kann.

3.3 Das Modul NEXUS

Der Server stellt die Verbindung zwischen den einzelnen Modulen her und sorgt für die Modulkonfiguration und -steuerung. Er bindet sich an einen festgelegten Port und erzeugt bei einem Verbindungseingang einen lokalen Serverprozeß, so daß für jede laufende Modulinstanz ein exklusiver Kommunikationskanal zur Verfügung steht.
Die Module binden sich beim Start an einen eigenen, lokalen Port und warten auf Befehle. Für jeden Befehl wird eine eigene Verbindung auf- und wieder abgebaut. Dies hat den Vorteil, daßmehrere CONTROL-Module von verschiedenen Stellen auf dasselbe Modul zugreifen können. Das Lauschen am Port ist nicht blockierend, so daß die Module im Hintergrund weiterlaufen können (Rechnung über Nacht).
Der Server kennt zwei Arten von Modulen, *legacy* und *nonlegacy*-Module. Die *legacy*-Module sind existierende Programme (Numerik, Graphik, Pre- oder Postprocessing), die mit minimalem Aufwand in die DEVISOR-Umgebung integriert werden sollen. Der Server übernimmt in

diesem Fall einen Großteil der Protokollbearbeitung und Konfiguration (Erzeugung der Konfigurationsdateien und ähnliches). Die Steuerung erfolgt über ein einfaches lokales Kommunikationsprotokoll. Dazu kann das Modul Steuerbefehle implementieren, welche im LOCUTUS-Linkmodul definiert werden. Ein Beispiel zur Integration eines Moduls ist im Anhang angegeben. Im einfachsten Fall wird aber das Modul nur gestartet und beendet, so daß in diesem Fall keine Änderungen am Programmcode notwendig sind.
nonlegacy-Module sind Module, die das komplette DEVISOR-Protokoll sprechen und ihre Konfigurations- und Steuerungsfunktionalitäten selbst regeln. Eine Beispielimplementierung ist im Anhang angegeben.
Um den Überblick zu behalten und das spätere Wiederaufsetzen einer Verbindung oder das spätere Aufschalten auf eine bestehende Rechung zu ermöglichen, führt der Server eine Routingtabelle, in der die Verbindung zwischen Modulname, Benutzername, ProjektID und lokalem Prozeß gespeichert ist.
Die Routingtabelle wird in regelmäßigen Abständen in eine Datei gesichert, um im Falle eines Serverfehlers oder Neukonfiguartion bestehende Rechnungen nicht zu verlieren..
Die Routingtabelle hat folgende Struktur:

```
typedef struct
{
  int entryfree;              /* Eintrag frei ja/nein */
  int moduleid;               /* Module-ID */
  int problemid;              /* Problem-ID */
  int instancehandle;         /* Instancehandle der
                                 zug.Applikation */
  int localport;              /* lokaler Port der
                                 zug.Applikation */
  char appl[512];             /* Pfad zur Applikation */
  char initappl[512];         /* Pfad zum
                                 Initialisierungsskript */
  char wdpath[512];           /* Pfad des
                                 Projektverzeichnisses */
  char projectid[128];        /* Projekt-ID in Stringform */
  char username[128];         /* Name des Benutzers */
  char display[128];          /* Bildschirmkennung
                                 in X11-Notation */
  int modus;                  /* Modus */
  int status;                 /* Status */
  int firstconfigured;        /* Wurde die Applikation schon
                                 konfiguriert ja/nein */
  struct setpar* apara;       /* Verweis auf die aktuellen
                                 Parameterwerte */
  struct tresult* result;     /* Verweis auf die
                                 Ergebnisliste */
  struct tstat* stat;         /* Verweis auf die
                                 Statistikenliste */
} routertableentry;
```

Codebeispiel 2: Routingtabelle

Die Information, welche Module auf einem Rechner verfügbar sind, sind in einer Konfigurati-

onsdatei `modules.conf` gespeichert. Der Inhalt dieser Datei wird anfragenden CONTROL-Modulen bei entsprechendem Protokollbefehl zurückgegeben.

Für jedes Modul wird ein Initialisierungsskript angegeben, welches die grundlegenden Initialisierung vornimmt wie Verzeichnisse erzeugen, Programmdateien kopieren etc. Ein zweites Skript startet schließlich die eigentliche Applikation. Die Skripte werden bei den entsprechenden Protokollbefehlen aufgerufen.

Viele, gerade ältere, Numerikprogramme erwarten einige ihrer Parameter als ausführbaren Code. Der Server bietet daher die Möglichkeit, aus den FunctionParametern ausführbaren Code zu erzeugen. Dieser Code wird von einem Zwischenmodul namens VINCULUM übersetzt. War die Kompilation erfolgreich, überlädt sich das Modul mit dem erzeugten Kompilat und führt so das eigentliche Modul aus.

Nach dem Starten forkt sich der Serverprozeß und bildet damit einen exklusiven Kommunikationskanal für das gestartete Modul. Für jedes CONTROL-Modul existiert ein eigener Serverprozeß. Für jede Kommandoübermittlung zum Modul wird jedoch eine eigene Verbindung auf und nach Kommando- und Antwortübermittlung wieder abgebaut, damit mehrere CONTROL-Module Zugriff auf das Modul haben können. Informationen der Routingtabelle wie Parameterwerte, Zeitschrittwerte, etc. werden vom geforkten Serverprozeß an den Ursprungsserver übertragen, damit die Informationen erhalten bleiben. Wird das CONTROL-Modul beendet, beendet sich auch der zugehörige Serverprozeß.

3.4 Das Modul CONTROL

Mit Hilfe des CONTROL-Moduls ist es möglich, komplette Simulationsabläufe durchzuführen, da im CONTROL-Modul gewissermaßen alle für die Durchführung einer Simulation benötigten Module „unter einen Hut“ gebracht werden. Dazu wird eine Benutzeroberfläche zur Verfügung gestellt, mit der es möglich ist, alle anderen Module zentral zu steuern und zu konfigurieren. Da für den kompletten Durchlauf einer Simulation in der Regel mehrere Module (für einen klassischen Ablauf siehe Kap. A.3) benötigt werden, werden diese komplett in Projekten verwaltet.

3.4.1 Funktionalität des Moduls

In CONTROL werden alle Module in Projekten verwaltet. Dies hat den Vorteil, daß die verschiedenen Module, die für die Durchführung einer vollständigen Simulation benötigt werden, zusammen verwaltet werden (geladen, gespeichert, konfiguriert, ...) und ein Datenaustausch nur zwischen Modulen des gleichen Projektes stattfinden kann. Auch werden alle Daten eines Projektes in einer eigenen Verzeichnisstruktur verwaltet. Insbesondere gilt dies für benutzerunabhängige Module wie solche aus der Numerik, die – einmal konfiguriert – nur ”rechnen”, und solche wie GRID, die über eine GUI interaktiv vom Benutzer bedient werden.

Wie auch in NET gilt auch in CONTROL: Alle Module werden komplett gleich behandelt. Die einzige Ausnahme ist, daß der Übersichtlichkeit halber Module nach Modultypen sortiert werden, die angebotene Funktionalität ist jedoch bei allen Modulen gleich. Ein typischer Ablauf für die Benutzung eines Moduls sieht folgendermaßen aus:

- Als erstes wird ein Server aus einer Serverliste ausgewählt (falls nötig, muß der neue Server zuerst eingetragen werden). Der Server liefert eine Liste mit allen auf diesem Server zur Verfügung stehenden Modulen zurück, aus denen das gewünschte Modul ausgewählt werden kann.
- Nach Start des Moduls kann nun jederzeit zu dem ausgewählten Modul Verbindung aufgenommen und diese auch wieder unterbrochen werden.
- Bevor das Modul jedoch rechnet, muß es zuerst konfiguriert werden. Da die Konfigurationsbeschreibung komplett vom Modul vorgegeben wird, wird in CONTROL dynamisch eine Benutzeroberfläche für die Konfigurationsmöglichkeiten erzeugt. Die Konfiguration erfolgt in drei Schritten: zuerst wird ein Problem aus einer Liste von Problemen ausgewählt, die das Modul zur Verfügung stellt, für das ausgewählte Problem wird eine Parameterliste angefordert, die konfiguriert werden kann, und zum Schluß erfolgt eine allgemeine Konfiguration des Moduls über eine Konfigurationsliste.
- Nach erfolgreicher Konfiguration steht eine Kassettenrekorderfunktion (Start, Stop, Pause, Rewind, Forward,...) zur Verfügung und mittels „Start“ kann das Modul zum Laufen gebracht werden. Inwieweit der Rest der Kassettenrekorderfunktion genutzt werden kann, hängt jeweils vom ausgewählten Modul ab. Rewind und Forward von Zeitschritten stehen in der Regel nur bei einem Numerikmodul zur Verfügung. Falls ein Modul den Pause-Modus unterstützt (z.B. Numerikmodule), kann es auch im laufenden Zustand neu konfiguriert werden, wobei jedoch in der dynamischen GUI auch nicht rekonfigurierbare Parameter geben kann, d.h. deren Werte können zwar eingesehen, jedoch nicht neu konfiguriert werden, solange das Modul läuft.
- Falls die Konfiguration Statistiken unterstützt, kann man sich Statistikdiagramme, z.B. über den Verlauf der Berechnung eines Numerikmoduls, anzeigen lassen.
- Falls Ergebnisse bei einem Modul zur Verfügung stehen, kann man diese herunterladen und im Projekt speichern.

3.4.2 Kommunikation mit anderen Modulen

Die Kommunikation mit allen anderen Modulen erfolgt komplett über den **ControlWrapper**, der von NET zur Verfügung gestellt wird (für eine Übersicht über die angebotenen Methoden siehe Kap. 3.2).

3.4.3 Paketstruktur

Das Modul CONTROL besteht aus folgenden Paketen:

devisor.control.app verwaltet alle Server und geladenen Projekte sowie die Anbindung zum NET-Modul.

devisor.control.gui beinhaltet alle Klassen, die zur Darstellung der Benutzeroberfläche gehören.

devisor.control.gui.dynamicGui enthält alle Klassen, die für die Erzeugung der dynamischen GUI benötigt werden.

devisor.control.gui.properties hält das Interface und seine Implementierungen der Einstellpanele für den Properties-Dialog des DEVISOR-Hauptfensters.

devisor.control.gui.images enthält alle verwendeten Bilder und Icons.

devisor.control.event enthält die benötigten Events und Listener.

devisor.control.util beinhaltet alle Toolklassen, die nicht in die anderen Pakete gehören.

devisor.vision.stats enthält alle Klassen, die für die Erzeugung einer Statistik nötig sind. Hierbei handelt es sich zwar um ein VISION-Paket, es wird jedoch nur in CONTROL verwendet.

3.4.4 Das Paket app

In diesem Paket werden die Projekte, Server und der Wrapper verwaltet, womit hier die komplette Anbindung an das NET-Modul erfolgt. Die Klasse **Control** ist die Hauptklasse dieses Pakets. Sie verwaltet zum einen den **ControlWrapper** und überprüft die *inbox* des Wrappers, ob Statistiken, Fehler- oder Statusmeldungen enthalten sind und leitet diese an das entsprechende Projektfenster weiter. Desweiteren wird eine Serverliste verwaltet und Methoden zum Laden und Speichern der Serverliste zur Verfügung gestellt. Die Serverliste wird in dem lokalen Verzeichnis des Benutzers gespeichert. In der Serverliste, realisiert durch die Klasse **ServerList**, werden alle **Server** verwaltet, dazu gehört Host, Port, Alias, Serverbeschreibung, Benutzername und eine Liste mit allen auf diesem Server verfügbaren Modulen. Zum anderen wird hier eine Projektliste verwaltet, die alle geladenen Projekte enthält. Die Klasse **ProjectObjectList** enthält diese Liste, in der Projekte neu angelegt, gelöscht, geladen und gespeichert werden können. Die Klasse **ProjectObject** verwaltet ein Projekt und die dazugehörigen Modullisten, in denen alle Module eines Typs verwaltet werden. Eine Modulliste wird verwaltet in der Klasse **ModuleObjectList**, in der neue Module eingefügt und wieder gelöscht werden können. In der Klasse **ModuleObject** wird das Modul selbst verwaltet. Dazu gehören zum einen alle modulbezogenen Informationen wie das **Module** selbst, Zustand des Moduls, Downloadinfos, die aktuelle Konfiguration, Modultyp und Verzeichnis, zum anderen enthält die Klasse alle Methoden zur Steuerung und Konfiguration des Moduls über den **ControlWrapper**.

3.4.5 Das Paket gui

Gui beinhaltet alle statischen Oberflächenelemente des CONTROL-Paketes. Hauptfenster ist **DeViSoR**, in welchem alle Projekte in internen Frames dargestellt werden.

Alle **app**-Klassen werden hier mit einer grafischen Interaktionsdarstellung versehen. Außerdem befinden sich hier alle Dialoge zum Erstellen eines neuen Projektes (**JNewProjectDialog**), eines neuen Servers (**JNewServerDialog**), ein **JCheckConnectionDialog** zum Testen einer Verbindung zu einem Server, der **JResultDownloadDialog**, um Ergebnisse der einzelnen Module herunterzuladen, und **JImportModuleDialog** und **JOpenModuleDialog** zum Importieren von auf anderen Rechnern gestarteten bzw. Öffnen von geschlossenen, aber nicht beendeten Modulen.

3.4.6 Das Paket properties

Im **properties**-Paket befindet sich nur das **JPropertyPanel** Interface und Implementierungen dieses Interfaces. Diese Implementierungen werden in den **JDeViSoRPropertiesDialog** von DEVISOR zur Veränderung der Einstellungen eingebunden. **JPropertyPanel** stellt dabei sicher, daß die benötigten Methoden zum Darstellen und Aktualisieren der Properties zur Verfügung stehen.

3.4.7 Das Paket dynamicGui

Dieses Paket enthält die dynamische GUI, mit der es möglich ist, ein Modul zu konfigurieren. Die Konfiguration eines Moduls besteht in der Regel aus drei Schritten:

1. Mit der Methode **getProblemList** des **ControlWrapper** wird die Problemliste des Moduls geholt. Aus diese Liste von Problemen muß ein Problem ausgewählt werden.

2. Für das ausgewählte Problem wird mit der Methode **getParameterList** vom **ControlWrapper** die Parameterliste geholt. Diese kann dann konfiguriert werden.

3. Schließlich wird mit der Methode **getConfigurationList** vom **ControlWrapper** eine Parameterliste für die problemunabhängige Konfiguration geholt.

4. Nach Beendigung der Konfiguration wird mit der Methode **configureModule** des **ControlWrapper** das Modul konfiguriert.

Die Klasse **JModuleConfigurationDialog** ist die Hauptklasse der dynamischen GUI und erzeugt einen Dialog, mit dem die Durchführung eines kompletten Konfigurationsdurchlaufs möglich ist. Analog z.B. zu einem Installationsdialog wird hier die Konfiguration in mehreren Schritten durchgeführt. Je nachdem bei welchem Schritt man sich gerade befindet, wird das jeweilige Panel für den aktuellen Schritt gezeigt. Dabei kann man immer vor und zurück gehen und erst nach einem kompletten Durchlauf kann die Konfiguration abgeschlossen werden. Im ersten Schritt wird die Problemliste geholt, und alle darin enthaltenen Probleme in einer Tabelle aufgelistet. Dies geschieht in dem Panel **JSelectProblemPanel**, das in diesem Schritt gezeigt wird. Bei erstmaliger Konfiguration werden die Parameterliste und die Konfigurationsliste neu geholt. Bei erneuter Konfiguration hingegen erfolgt eine Abfrage, ob die alten Einstellungen übernommen werden sollen oder ob die Konfiguration komplett neu erfolgen soll, falls wieder das gleiche Problem gewählt wird. Im ersten Fall werden die aktuelle Parameter- und Konfigurationsliste übernommen, die in dem zu dem Modul gehörigen **ModuleObject** gespeichert sind, ansonsten werden eine neue Parameterliste und eine neue Konfigurationsliste vom Modul geholt. Im zweiten und dritten Schritt werden die Parameterliste und die Konfigurationsliste in einem **JDynamicGuiPanel** dargestellt. Genaugenommen handelt es sich um eine SplitPane, die auf der linken Seite einen Baum enthält, der die Struktur der dynamischen GUI enthält. Die gegebene Parameterliste stellt die Wurzel dar und alle weiteren Knoten repräsentieren enthaltene **NodeParameter**. Ein **NodeParameter** enthält selbst wieder Parameterlisten, wodurch sich die Baumstruktur ergibt. Alle weiteren **Parameter** einer Parameterliste, wobei es sich um die übergebene Parameterliste oder um eine Parameterliste eines **NodeParameter** handeln kann, stellen GUI-Elemente dar und werden in einem Panel zusammengefaßt. Für die Verwaltung der Knoten des Baums gibt es die Klasse **ParamTreeNode**. In ihr wird jeweils das Panel einer

Parameterliste verwaltet. Falls im Baum ein Knoten ausgewählt wird, wird auf der rechten Seite der SplitPane das entsprechende Panel angezeigt. Das Panel einer Parameterliste setzt sich wieder modular aus einzelnen Panels zusammen, die die GUI für die einzelnen **Parameter** der Parameterliste enthalten. Für alle möglichen **Parameter** der Parameterliste gibt es jeweils eine Klasse, die als Pendant ein passendes Panel erzeugt. Analog zur abstrakten Klasse **Parameter** erben die Panel-Klassen alle von der abstrakten Klasse **AbstractDynGuiPanel**. Sie enthält die abstrakten Methoden **moduleIsRunning**, **validInput** und **setAllParameters**. Die Methode **moduleIsRunning** wird im Konstruktor der jeweils erbenden Klasse aufgerufen und erhält einen *Boolean*-Wert übergeben, der anzeigt, ob das Modul sich in einem laufenden Zustand befindet. Falls dies der Fall ist und der gegebene Parameter nicht rekonfigurierbar ist, werden die Komponenten des Panels so „disabled“, daß zwar noch die Parameterwerte sichtbar, aber nicht veränderbar sind. Die Methode **validInput** liefert einen **Boolean**-Wert zurück, der anzeigt, ob die Eingabe in dem Panel ein zulässiges Format hat. Die Methode **setAllParameters** setzt die Eingabe aus dem GUI-Panel in dem dazugehörigen **Parameter**. Es gibt folgende Panel-Klassen:

- **JScalarPanel** für einen **ScalarParameter**. Die möglichen GUI-Typen sind *TextField*, *TextArea*, *Slider*, *CheckBox* oder *RadioButtons*.

- **JVectorPanel** für einen **VectorParameter**. Die möglichen GUI-Typen sind *TextFields*, *CheckBoxes*, *Sliders*, oder ein *ColorChooser*.

- **JFunctionPanel** für einen **FunctionParameter**. Die GUI hier besteht aus einer Tabelle für die Funktionsparameter, sowie einer *TextArea* für die Funktion selbst.

- **JGridPanel** für einen **GridParameter**. Hier kann eine FEAST-Datei mittels **FileChooser** ausgewählt und betrachtet werden. Zum Betrachten einer Domain gibt es die Klasse **JDomainViewerDialog**, die einen Dialog erzeugt, der ein 2D-ViewerPanel enthält.

- **JSelectionPanel** für einen **SelectionParameter**. Die möglichen GUI-Typen für die Auswahl von Parametern sind *RadioButtons* und *ComboBox* für die Auswahl eines einzigen Parameters, *CheckBoxes* für die Mehrfachauswahl und *Selection* für Mehrfachauswahl mit mehrfacher Wahlmöglichkeit des gleichen Parameters. Bei *Selection* hat man zwei Listen, eine, die die verfügbaren Parameter enthält, und eine zweite Liste, die die ausgewählten Parameter enthält. Die Konfiguration der jeweils ausgewählten Parameter erfolgt über weitere Dialoge. Falls es sich bei dem ausgewählten Parameter um einen **NodeParameter** handelt, wird der Dialog **JDynamicGuiDialog** aufgerufen, der wieder für die Parameterliste des **NodeParameter** ein **JDynamicGuiPanel** enthält. In allen anderen Fällen wird der Dialog **JSelectionGuiDialog** aufgerufen, der das entsprechende Panel für den Parameter enthält. Falls der gegebene **SelectionParameter** die ID 1 hat, handelt es sich um den Statistikparameter, der eine Ausnahme bildet. Bei ihm ist keine Konfiguration, sondern nur die Auswahl der Parameter möglich.

- **JDataSourcePanel** für einen **DataSourceParameter**. Hier kann von einem gewählten Numerikmodul ein Ergebnis ausgewählt und konfiguriert werden. In dem Dialog **JSelectTimeStepsDialog** ist es möglich, für ein gewähltes Ergebnis ein Datenfeld und Zeitschritte auszuwählen.

Aus diesen Panels setzt sich nun modular jeweils das Panel eines Knotens zusammen. Die Klassen **ParamTreeNode** und **JDynamicGuiPanel** besitzen jeweils die Methoden **validInput** und **setAllParameters**. Bei **validInput** wird hier *true* zurückgegeben, wenn die Eingaben für ein Knotenpanel oder für eine Parameterliste zulässig sind. Bei **setAllParameters** werden alle Parameter für ein Knotenpanel oder für eine Parameterliste gesetzt. Solange die Eingabe für das Panel des ausgewählten Knotens unzulässig ist, kann kein anderer Knoten gewählt werden bzw. in der Konfiguration einen Schritt weiter oder zurück gegangen werden.

3.4.8 Das Paket event

Im **event**-Paket liegen die Klassen der benutzten Events und die entsprechenden Listener. Als erstes **ServerChangedEvent** und **ServerListChangedEvent**, welche im Falle einer Veränderung eines Wertes eines Servers bzw. der Serverliste an alle **ServerChangedListener** bzw. alle **ServerListChangedListener** geliefert werden. Zum anderen **CloseTabEvent**, der von den **JCloseableTabbedPanes** an die **CloseTabListener** gefeuert wird, wenn sie geschlossen werden sollen.

3.4.9 Das Paket util

In diesem Paket sind die Werkzeugklassen, die nicht in ein anderes Paket paßten, definiert.

- **DeViSoRProperties** kapselt die Properties für DeViSoR als statische Klasse sowie die Laden- und Speichern-Funktionalität der Einstellungen.

- **DeViSoRResourceBundle** kapselt ähnlich wie DeViSoRProperties das Internationalisierungsresourcebundle für CONTROL.

- **Package** stellt Methoden zum Finden aller Klassen und Klassennamen eines Package im Classpath bereit.

- **SimpleFileFilter** ist ein Filter, sowohl für grafische, als auch textuelle Auflistungen von Dateien in einem Verzeichnis. **SimpleFileFilter** kann nach mehreren Dateiendungen oder einem Teil des Dateinames mit und ohne weitere Ausfilterung von Unterverzeichnisse filtern.

3.4.10 Das Paket stats

Dieses Paket enthält die Klassen, die für die Darstellung der statistischen Daten einer Berechnung genutzt werden. Es besteht im wesentlichen aus den folgenden Klassen:

- Die Klasse **StatManager** kapselt die Darstellung der Statistikdaten komplett nach außen, d.h. die übrigen Teile des Programmes kommunizieren nur mit dieser. Sie stellt Methoden zur Erstellung einer neuen Statistikdarstellung (inlusive der dazugehörigen Konfiguration) sowie zum Echtzeit-Update der Statistikdaten zur Verfügung.

- Eine Instanz der Klasse **StatWindow** stellt das Fenster dar, in dem die entsprechend konfigurierten Statistiken dargestellt werden, je nach Konfiguration werden so eine oder mehrere Statistiken angezeigt.

- Durch die Klasse **StatChart** wird jeweils eine einzige solcher Statistiken dargestellt, innerhalb dieser Klasse findet die eigentliche Visualisierung der Statistikdaten dar. Dazu wird eine Instanz aus einer der Klassen **LBChart2D**, **LLChart2D** oder **PieChart2D** (aus der benutzten Java Bibliothek Chart2D von Jason J. Simas) erzeugt und innerhalb eines **JPanels** graphisch dargestellt.
- Die Konfiguration der Statistikdarstellung wird durch die Klasse **StatConfigurationDialog** vorgenommen, die einen Dialog für die Konfiguration der einzelnen Statistikdarstellungen bietet.

Die verwendete Java Bibliothek Chart2D, Copyright (C) 2001 von Jason J. Simas, steht unter der GNU Lesser General Public License und kann somit in dem DEVISOR-Projekt frei eingesetzt werden.
Die Klasse **StatManager** kapselt, wie bereits zu Eingang erwähnt, die komplette Konfiguration und Darstellung der Statistiken, das CONTROL-Modul muß nur die Methoden dieser Klasse integrieren, um die volle Funktionalität der Statistikdarstellung anbieten zu können.
Eine neue Instanz dieser Klasse wird hierbei über den Konstruktur erstellt, der als Parameter zwingend einen **SelectionParameter** enthalten muß, in welchem die angebotenen Statistikdaten und deren Format beschrieben werden, wie das auch bei den für die anderen Module benutzen Parameterlisten der Fall ist. Durch Aufruf des Konstruktors wird gleichzeitig der Konfigurationsdialog dargestellt, in welchem die Art, Anzahl und Konfiguration der Statistikdiagramme bestimmt werden können. Nach Abschluss der Konfiguration wird ein neues Fenster erzeugt, das zunächst nur eine oder mehrere leere Statistikdiagramme enthält. Diese werden nun durch das CONTROL-Modul durch den Aufruf der **update()**-Methode des **StatManager**s mit Daten gefüllt, und stellen diese je nach Konfigurationseinstellungen dar. Die **update()**-Methode nimmt hierbei entweder eine Instanz der Klasse **Statistics** oder **StatisticsList**, letztere enthält sinnvollerweise eine Liste an Statistics-Objekten. Durch jedes Update werden die neu hinzugefügten Daten direkt und ohne Verzögerung dargestellt. Ein Schließen des Fensters versteckt dieses, die Datenstruktur bleibt jedoch im Speicher erhalten, und weitere Updates werden weiterhin (ohne Darstellung) eingearbeitet. Ein Aufruf der Methode **show()** bringt das versteckte Fenster wieder zur Anzeige.

3.5 Das Modul GRID

Das Modul GRID stellt dem Benutzer einen Editor für zwei- und dreidimensionale Gitter zur Verfügung. Ein Gitter beschreibt den Rechenbereich einer Simulation und die kleinsten Recheneinheiten – die sogenannten Finiten Elemente.Da die Anzahl der Gitterelemente in einer komplexen Rechnung eine Million übersteigen kann, dient GRID als Editor für Grobgitter, die dann während der Berechnung verfeinert werden (Kap. 2.4).
Das Modul besteht aus zwei Teilen. Zum einen gibt es den Framework-Teil, der die Datenstrukturen und die GUI-Komponenten enthält, die zur Beschreibung und Darstellung des Gitters dienen, und zum anderen den eigentlichen GRID-Editor, der es dem Benutzer ermöglicht, durch Maus- und Tastatureingaben ein Gitter zu erzeugen.

3.5.1 Funktionalität des Moduls

- Erstellung und Bearbeitung zweidimensionaler Gitter,

- Erstellung und Berarbeitung dreidimensionaler Gitter,
- die Möglichkeit, die Gitter im FEAST- bzw. FEAST3D-Format abzuspeichern und zu laden,
- Importieren von LightWave-Dateien als Teil eines Gitters und
- Übermittlung der erstellten Gitter über das Netzmodul.

3.5.2 Kommunikation mit CONTROL über das NET-Modul

GridNet übernimmt die Kommunikation mit der Netzwerkschnittstelle durch den **GridWrapper**. Über das Netzwerk kann ein Gitter übermittelt werden, das der Editor selbständig lädt und sich abhängig vom Gittertyp (2D/3D) in den entsprechenden Modus schaltet. **GridNet** implementiert dazu das Interface **Wrappable** aus dem NET-Paket.
Wurde ein Gitter gespeichert, so gibt GRID auf Anfrage von CONTROL eine Liste der gespeicherten Dateien zurück. Die Übertragung der Dateien funktioniert wie ein Ergebnis-Download bei den anderen Modulen.

3.5.3 Das Paket Framework

Das Framework-Paket besteht aus zwei Unterpaketen. Im Foundation-Paket befinden sich die Datenstrukturen, in denen die Gitterdaten abgespeichert werden. Im Viewer-Paket befinden sich Klassen, die dazu dienen, das Gitter zu visualisieren.

Das Paket Foundation

Ein Gitter enthält drei unterschiedliche Typen von Objekten:

- Boundaries beschreiben den Rechenbereich. In einer Strömungssimulation kann dies z.B. ein Kanal oder der Bereich außerhalb der Karosserie eines Autos sein. Die Datenstrukturen des Frameworks erlauben es, zweidimensionale Boundaries durch Kreise, Linien, und Kombinationen dieser beiden, und dreidimensionale Boundaries durch Kugeln, Quader und Dreiecksnetze darzustellen.

  Die Datenstrukturen, die dies leisten, sind **BoundaryCircle**, **BoundarySegmentList** im zweidimensionalen und **BoundarySphere**, **BoundaryCubicle** und **BoundaryTriangulation** im dreidimensionalen Fall.
- Finite Elemente beschreiben die kleinsten Recheneinheiten. Dies können in der Ebene Dreiecke oder Vierecke (Tris oder Quads) sein und im Raum Tetraeder oder Hexaeder (Tetras oder Hexas).
- Desweiteren gibt es die sogenannten Basisobjekte, aus denen sich die Boundaries und die Finiten Elemente zusammensetzen. Dadurch ist es möglich, die Nachbarschaftsbeziehungen zwischen den Objekten ebenfalls in der Datenstruktur festzuhalten.

Jedes Gitter-Objekt – egal ob Boundary, Finites Element oder Basisobjekt – enthält eine Liste seiner Eltern. In dieser Liste stehen die Objekte, die das Objekt, in dessen Liste sie stehen, benutzen. Eine Kante steht beispielsweise in der Elternliste ihres Start- und Endknotens. Stehen

zwei verschiedene Kanten in der Elternliste eines Knotens, bedeutet dies, daß diese Kanten benachbart sind. Analog sind zwei Facetten (3D) oder Tris (2D) benachbart, wenn sie zusammen in der Elternliste einer Kante stehen. Der **java.util.Vector**, der die Eltern enthält, kann mit der Methode **getParents()** abgefragt werden.
Außerdem erhält jedes Objekt eine für seinen Typ eindeutige Nummer.
Es folgt eine kurze Erläuterung, welche Information die einzelnen Klassen speichern (die Informationen über die Eltern sind in jedem Objekt vorhanden):

- Basis-Objekte:
  - **ElementNode** enthält die Koordinaten eines Knotens.
  - **Edge** enthält Referenzen auf zwei Knoten, die den Start- und den Endpunkt der Kante beschreiben.
- Finite Elemente:
  - **Face** ist eine abstrakte Klasse, die eine Facette beschreibt. Sie besteht aus einer festen Anzahl von Kanten.
  - **Face2D** ist ebenfalls eine abstrakte Klasse. Diese beschreibt eine Facette in der Ebene. Diese Klasse enthält zusätzliche Informationen, die es ermöglichen, die Facette in der Ebene weiter zu verfeinern.
  - **Tri** ist eine Facette, die aus drei Kanten besteht, im geometrischen Sinne also ein Dreieck.
  - **Quad** ist eine Facette, die aus vier Kanten besteht. Im geometrischen Sinne stellt sie ein Viereck dar.
  - **Body** stellt einen Körper im Raum dar. Er kann aus einer festen Anzahl von Seitenflächen (**Face**) bestehen und enthält zusätzlich Informationen, wie der Körper im Raum weiter verfeinert werden kann.
  - **Tetra** ist ein Körper, der aus vier Seitenflächen besteht (z.B. eine dreiseitige Pyramide).
  - **Hexa** ist ein Körper, der aus sechs Seitenflächen besteht (z.B. ein Würfel).
- 2D-Boundaries:
  - **BoundaryBase** ist die Basisklasse für alle Boundaries.
  - **BoundaryCircle** beschreibt einen kreisförmigen Bereich in der Ebene. Der Mittelpunkt des Kreises wird durch einen Knoten beschrieben. Der Radius wird als Double-Wert gespeichert. Ein Kreisausschnitt wird ebenfalls durch diese Klasse beschrieben, wobei Start- und Endwinkel als Double gespeichert und als Winkel im Bogenmaß interpretiert werden. Kreissegmente sind jedoch nur als Teil einer **BoundarySegmentList** zulässig. Ist der Endwinkel größer als der Startwinkel, so ist die Umlaufrichtung des Kreises gegen den Uhrzeigersinn. Dies wird durch FEAST so interpretiert, daß der Kreisbogen eine Fläche umschließt. Ist die Umlaufrichtung gegen den Uhrzeigersinn, so beschreibt die Datenstruktur die Fläche außerhalb des Kreises. Der Winkel 0 liegt auf 3 Uhr-Position.

  - **BoundarySegmentList** kann aus Kanten und Kreissegmenten bestehen. Diese werden in einer Liste abgespeichert (Java-Datenstruktur **Vector**). Die Reihenfolge in der Liste spiegelt die Reihenfolge der Segmente wieder und gibt damit die Umlaufrichtung der **BoundarySegmentList** an. Auch hier gilt: sind die Segmente im Uhrzeigersinn angeordnet, so beschreibt diese Klasse den inneren Bereich und ansonsten den äußeren Bereich.

- 3D-Boundaries:

  - **BoundarySphere** beschreibt eine Kugel im Raum. Der Mittelpunkt der Kugel ist ein Knoten, der Radius wird als Double gespeichert.

  - **BoundaryCubicle** beschreibt einen Quader im Raum. Er wird durch die Angabe von acht Knoten beschrieben. Die Bedingung, daß nur rechte Winkel erlaubt sind, ist nicht in der Datenstruktur verankert, sondern wird nur beim Erstellen im Editor erzwungen.

  - **BoundaryTriangulation** beschreibt einen Körper, der durch Dreiecke definiert wird. Dazu wird die Hilfsdatenstruktur **Triangle** benutzt. Die Instanzen von Triangle werden in einem **Vector** abgelegt. In der Triangle-Klasse werden zusätzlich zu den drei Knoten, die die Eckpunkte des Dreiecks bilden, Referenzen auf die direkt benachbarten Dreiecke abgespeichert. Die Klasse **devisor.framework.-toolbox.GeometryExtractor** erzeugt aus einem Java3D-Szenegraphen eine **BoundaryTriangulation**. Da der Szenegraph jedoch redundante Dreiecke enthält, gehen diese auch in die Boundary-Datenstruktur mit ein. Ursprünglich war geplant, die Triangulation aus einer vorgegebenen Punktemenge zu erzeugen. Dies war jedoch im Rahmen der Projektgruppe nicht mehr zu verwirklichen.

Zwei Klassen, die in der bisherigen Beschreibung ausgelassen wurden, sind **BoundaryNode** und **BoundaryNode3D**. Diese dienen der Angabe von relativen Positionen der Knoten. Sie besitzen jedoch die gleiche Funktionalität wie **ElementNode**s.

In einem **BoundaryNode** sind die Nummer eines 2D-Boundaries (dies sind Boundaries, die das Interface **Segment** implementieren) und ein Parameter t gespeichert. Ist die Boundary beispielsweise ein Kreis und der Parameter 0.5, so liegt der **BoundaryNode** an der 12 Uhr-Position auf dem Kreis. Für **BoundarySegmentList**s (diese setzen sich aus Kanten und Kreissegmenten zusammen – entspricht der Parameterwert 2.5, der Hälfte der 3. Segments.

BoundaryNode3Ds können auf Boundaries liegen, die das Interface **Segment3D** implementieren. Außer der Boundary-Nummer speichern sie zwei Winkel α und β und einen Integer-Wert n. Um die Position des **BoundaryNode3D** zu berechnen, rotiert man einen Strahl mit Richtungsvektor $(1, 0, 0)^t$ zuerst um α um die Z-Achse und dann um den Winkel β um die X-Achse. Der Startpunkt des resultierenden Strahls wird in den Schwerpunkt der Boundary verschoben. Der n-te Schnittpunkt des Strahls mit dem Boundary gibt die Position des **BoundaryNode3D**. Der Schwerpunkt ist der Mittelpunkt der Boundingbox der Boundary. Da die Implementation von **Segment3d** auch die Implementation der Methode **getBoundingBox()** erzwingt, kann dieser leicht ermittelt werden.

```
/* Strahl vor der Rotation */
Point3d p = new Point3d(1.0, 0.0, 0.0);

/* Rotationsmatrizen */
Matrix3d M = new Matrix3d(), N = new Matrix3d();
M.rotX(beta);
N.rotZ(alpha);
M.mul(N);

/* Durchfuehrung der Rotation,
 * verschieben in den Schwerpunkt */
M.transform(p);
p.add(center);
```

Codebeispiel 3: Berechnung des Strahls

Es fehlen die Klassen **Boundary** und **Domain**. Diese sind intelligente Container für die Boundary-Objekte, die Finiten Elemente und ihre Bestandteile (Knoten, Kanten, usw.). Die Aufteilung in Boundary und Domain ist prinzipiell unnötig. Sie ist durch Mißverständnisse während des Entwurfs entstanden. **Domain** und **Boundary** enthalten **createXYZ(...)-**, **addXYZ(...)-** und **removeXYZ()**-Methoden. Die **add-** und **remove**-Methoden dienen dazu, die Objekte in der Instanz zu speichern und die Listen der Eltern aktuell zu halten. Die **createXYZ**-Methoden erzeugen eine neue Instanz von **XYZ** und weisen diesem eine eindeutige Nummer zu. Außerdem werden Plausibilitätsüberprüfungen durchgeführt (z.B. sind keine Kanten zugelassen, die aus zweimal demselben Knoten bestehen). Werden diese nicht erfüllt, so werden entsprechende Exceptions geworfen.
Zusätzlich gibt es eine Vielzahl von Methoden, um Objekte nach bestimmten Kriterien zu suchen, z.B. **findEdgeByNumber(3)** um die Kante 3 zu suchen, oder **findEps(p, v, d, bb)** um einen Knoten zu finden, der höchstens Abstand d von der Gerade $\mathsf{p} + alpha \cdot \mathsf{v}$ hat und innerhalb der Boundingbox bb liegt.

Das Paket devisor.framework.viewer

In diesem Paket werden die Methoden und graphische Schnittstellen bereitgestellt, um die im **devisor.framework.foundation**-Paket vorgestellten Objekte zu visualisieren. Dabei sind natürlich nicht nur die Objekte gemeint, sondern die komplette Visualisierungsumgebung, die es erlaubt, die drei Fenster und die Koordinatenachsen darzustellen und die absoluten Koordinaten der Rohdaten in der Domain in relative Koordinaten der Ansichtsfenster umzurechnen.
Dieses Paket stellt die Schnittstelle zwischen den Elementen der Domain und dem Editor bereit. Auch wenn es möglich ist, die **ViewerPanel**s ohne den Editor zu benutzen, machen manche Methoden nur Sinn, wenn man sie aus Sicht des Editors benutzt (z.B. **setSelection(...)**).
Die wichtigste Klasse ist **DrawingArea**. Sie definiert die Zeichenfläche mit den ihr zugehörigen Komponenten. Sie enthält eine Instanz der Klasse **Domain** und eine Instanz einer Unterklasse der Klasse **ViewerPanel** als Hauptattribute. Somit wird die Verbindung zwischen den *entity*- und *view*-Datenstrukturen hergestellt. Natürlich enthält sie auch die Optionen, die die Darstellung benutzergerecht modifizieren, die Objekte z.B. in einem bestimmten Grade transparent darstellen, etc.

Die Aufgabe dieser Klasse ist, zu bestimmen, welche Elemente der Domain in der zugewiesenen ViewerPanel aktuell sichtbar sind, um sie anschließend korrekt bezogen auf die relativen Koordinaten des Viewer-Fensters und den Zoom-Faktor zu zeichnen.
Es werden ein Gitter(**drawGrid**) und die Koordinatenachsen(**drawAxes**) im Hintergrund gezeichnet, um dem Benutzer das maßstabgerechte Eingeben der Elemente zu erleichtern. Die Achsen werden in der gleichen Farbe eingefärbt, wie die Zahlen in dem Koordinatenfenster, und es werden je nach Typ des ViewerPanels (Top, Front oder Side) die richtigen Achsen gesetzt. Dann kommt die Domain an die Reihe (**drawDomain**, **drawBoundary**), so daß alle vorhandenen sichtbaren Elemente auch wirklich richtig gezeichnet werden - abhängig davon, ob die Elemente zu dem aktuellen Modus, 2D oder 3D passen oder nicht. Zusätzlich wird der POI (*point of interest*) gezeichnet(**drawPOI**), welcher den Mittelpunkt des aktuellen Ansichtsfensters bestimmt. Die Elemente (Knoten) in der Auswahl werden gesondert gezeichnet(**drawSelection**) und auch die ViewBox markiert hier durch einen rechteckigen Rahmen die vom Benutzer gesetzten Grenzen der Ansicht(**drawViewBox**). Die Methode **draw** initiiert das Zeichnen all dieser Elemente.

Die abstrakte Klasse **ViewerPanel** und ihre Unterklassen **ViewerPanel2D** und **ViewerPanel3D** beschreiben das eigentliche Fenster, in das Instanzen von DrawingArea die Objekte malen. Dieses besitzt Attribute, um sich selbst als ein bestimmtes Fenster(Top, Front oder Side) mit bestimmten Eigenschaften (z.B. ob es das aktive Fenster ist) identifizieren zu können.
Es enthält Methoden zur Festlegung des aktuellen Maßstabes (z.B. **zoomToFit**).

3.5.4 Das Paket devisor.grid

Dieses Paket besteht aus einer Anzahl von Unterpaketen, die dafür verantwortlich sind, die grafische Schnittstelle zum Benutzer zu realisieren. Damit sind vor allem das Hauptfenster, die Dialoge, die Ereignisbehandlung, die Hilfe-Datenbank und die Benutzer-Optionen-Datenstrukturen gemeint.

Am wichtigsten ist das Paket **devisor.grid.main**, das die Startklasse **GridApp** und die Hauptfensterklasse **GridMainFrame** enthält. Mit GridApp wird das Programm gestartet und es werden Einstellungen aus einer Konfigurationsdatei, oder, falls nicht vorhanden[1], hartkodiert übernommen. **GridMainFrame** enthält Instanzen von fast allen Klassen von **devisor.framework** und **devisor.grid** und verwaltet diese. Sie kann als die Kernklasse des GRID-Pakets bezeichnet werden. Sie ist die Kontrollklasse, die sowohl über die Schnittstelle zum Benutzer als auch über die Zuweisung an die **Entity**-Klassen verfügt. Die Funktionalität wird über weitere Klassen abgebildet, z.B. werden die Elemente über Dialoge oder **MouseListener** gesetzt, die aber Attribute von **GridMainFrame** sind und dadurch direkt mit aktualisiert werden, wenn sich etwas an den internen Datenstrukturen ändert. Für viele der Knöpfe mit einer simplen Funktionalität, wie z.B. „Beenden“ wurde keine explizite Event-Handler-Klasse geschrieben, sondern jeweils eine interne Klasse vom Typ **ActionListener** definiert, in der dann die Event-Behandlung stattfindet. **GridMainFrame** enthält vier Instanzen der Subklassen des **devisor.framework.-viewer.ViewerPanel**. Auf diesen wird dann aus verschiedenen Sichten die Domain dargestellt.

[1]Beim ersten Start kann es sehr wohl sein, daß die Konfigurationsdatei noch nicht erstellt worden ist. Sie wird aber nach dem ersten Start automatisch für das System, auf dem es läuft, erstellt und konfiguriert.

Das letzte Panel ist das 3D-Panel, das eine dreidimensionale Sicht auf die Objekte erlaubt. Alle Operationen in GRID gehen über **GridMainFrame**.
Zusätzlich befinden sich dort die Verbindung zu der Netzschnittstelle, die Klasse **GridNet**. Diese implementiert das Interface **Wrappable** und enthält eine Instanz der Klasse **GridWrapper** als Attribut, über den dann über das Netz kommuniziert werden kann (Siehe 3.2).
Auch die Schnittstellen zu den externen Datentypen, wie Java-Wavefront-Dateien (**WaveFront-Importer**) und zu dem Datentyp FEAST (**FeastLoader**), unter welchem alles, was mit GRID erstellt wird, abgespeichert wird, befinden sich in diesem Paket.
Eine Hilfsklasse, der **MouseModeListener**, ist ebenfalls in diesem Paket zu finden. Sie schaltet in **GridMainFrame** zwischen verschiedenen Mouse-Modi um. Mouse-Modi sind Konstanten, die bestimmen, welcher Event-Handler intern aktiv sein soll.

Im Paket **devisor.grid.dialogs** befinden sich alle Dialoge von GRID, deren Namen gleichzeitig auch ihre Funktionalität beschreiben. Es handelt sich um

AboutDialog,	**HierarchyDialog**,
BoundaryCubicleDialog,	**NodeDialog**,
BoundaryDeleteDialog,	**OptionsDialog**,
BoundaryNode3DDialog,	**PasteAtDialog**,
BoundaryNodeDialog,	**PointOfInterestDialog**,
CircleDialog,	**QuadDialog**,
CylinderDialog,	**SphereDialog**,
EdgeDialog,	**TetraDialog**,
ExceptionDialog,	**TriDialog**,
GridSplashDialog[2],	**ViewBoxDialog**,
HelpDialog,	**VisibilityDialog**.
HexaDialog,	

Die meisten dieser Dialoge sind gleichartig aufgebaut und unterscheiden sich nur in ihrer Funktionalität. Die meisten sind dazu da, dem Benutzer die Möglichkeit zu geben, neue Elemente in die Datenstruktur und auf die Zeichenfläche einzufügen. Um noch mehr über die Funktionalität der Dialoge zu erfahren, siehe das Handbuch von GRID in Kapitel A.5.
Hier wird exemplarisch der **NodeDialog** vorgestellt, um die Funktionalität, die allen Dialogen zugrunde liegt, vorzustellen.
Der Konstruktor von **NodeDialog** ruft die Methode **createDialog** auf, mit der der Dialog gezeichnet wird. Es wird eine **JComboBox** für die Auswahl der schon vorhandenen Knoten, drei **JTextPane**s zur Eingabe der Koordinaten und vier **JButton**s, [OK], [APPLY], [CANCEL] und [DELETE] dargestellt. Die Methoden **update**, **updateCombo** und **selectionChanged** sorgen dafür, daß die ComboBox mit den vorhandenen Knoten stets aktuell bleibt, auch wenn Knoten außerhalb von dem Dialog erstellt werden (z.B. mit dem **AddNodeListener**). Die Methode **assignNode** synchronisiert die Anzeige mit dem übergebenen Knoten. Sie wird beim Erstellen eines neuen Knotens aufgerufen. Am wichtigsten ist die Methode **actionPerformed**, in der die Ereignisse behandelt werden, wenn einer der Buttons gedrückt wird. Beim Button [OK] werden die Einstellungen übernommen und der Dialog wird geschlossen, [APPLY] macht das gleiche, ohne den Dialog zu schließen, [DELETE] löscht den aktuell ausgewählten Knoten und [CANCEL] verläßt den Dialog ohne Änderungen.

[2]Dies ist eigentlich kein Dialog, sondern das Bild mit einem Fortschrittsbalken, das beim Starten des Programms erscheint. Hier wurde er nur der Vollständigkeit halber angegeben.

Die Dialoge werden in **devisor.grid.main.GridMainFrame** eingetragen und global aktualisiert. In der Methode **createDialogs** werden sie erzeugt. Die Methode **update** aktualisiert alle (sichtbaren) Dialoge auf alle Veränderungen in der Domain. Ansonsten werden die Dialoge als Listener an die Menüs und Werkzeugbuttons angehängt. Die genauere Beschreibung dieses Vorgangs steht in B.4.

Das Paket **devisor.grid.event** enthält alle Event-Behandlungsroutinen, die man bei der Eingabe von Elementen und bei diversen anderen Mouse-Optionen braucht. Sie sind genauso wie die Dialoge selbsterklärend. Es sind im einzelnen:

AddCircleListener,	**MoveSelectionListener**,
AddCubicleListener,	**MoveViewListener**,
AddCylinderListener,	**NewProjectListener**,
AddEdgeListener,	**RotateSelectionListener**,
AddNodeListener,	**ScaleSelectionListener**,
AddQuadListener,	**SelectBoxListener**,
AddSegmentListListener,	**ShowDialogListener**,
AddSphereListener,	**ShowHierarchyListener**,
AddTriListener,	**ShowPointOfInterestDialogListener**,
DeleteSelectionListener,	**TriangulateListener**,
ExitListener,	**ViewBoxDialogListener**,
GridKeyListener,	**ZoomFactorListener**.

Um die genaue Funktionalität dieser Klassen zu erfahren, wird wieder auf das Handbuch von GRID, A.5 und die API-Dokumentation verwiesen.

An dieser Stelle wird exemplarisch auf die Klasse **AddNodeListener** eingegangen. Die anderen Listener unterscheiden sich nur in der spezifischen Funktionalität der Methoden.

Die wichtige Methode dieser Klasse ist **mouseClicked**. Diese fängt das Ereignis des Mausklicks ab und erzeugt daraufhin einen neuen Knoten in der Domain, die in der übergebenen **GridMainFrame**-Instanz enthalten ist. Dann wird der Knoten in alle Instanzen von **framework.viewer.DrawingArea** gezeichnet, bezogen auf seine neuen fenster-relativen Koordinaten. Die Koordinaten des neu erzeugten Knotens hängen auch von dem eingestellten Mouse-Snapping-Wert ab, der in der Routine **snap** berechnet wird.

In **devisor.grid.main.GridMainFrame** werden die Listener immer dann aufgerufen, wenn auf eines der Werkzeugbuttons (z.B. Add Node) mit der Maus geklickt wird. Sie sind die Ereignisbehandlung der Buttons.

Die im Paket **devisor.grid.backend** sich befindende Klasse **GridManager** setzt wichtige Programmkonstanten und Programmodis. Die andere Klasse **StatusBar** realisiert einen Fortschrittsbalken, der z.B. beim Laden erscheint.

Das Paket **devisor.grid.info** enthält keine Klassen, sondern die Inhalte für den About-Dialog: Autorenvorstellung, GPL-Hinweise, Danksagungen, etc. Zusätzlich enthält dieses ein Unterpaket **manual**, in dem das Grid Handbuch(A.5) im HTML-Format enthalten ist.

Zusätzlich existiert das Verzeichnis **devisor.grid.images** in der Paketstruktur, das die Icons, welche in dem Programm verwendet werden, enthält. Die Bilder, die die Dialoge verwenden, befinden sich meist in dem Paket, welches den gewünschten Dialog enthält.

3.6 Das Modul VISION

Das VISION-Modul wird innerhalb des DEVISORProjekts zur Visualisierung der aufkommenden Daten benötigt. Um die Visualisierung einer großen Anzahl verschiedener Simulationstypen zu unterstützen, werden eine ganze Reihe unterschiedlicher Visualisierungstechniken angeboten.

3.6.1 Funktionalität des Moduls

- Kommunikation mit den anderen Teilen über **VisionControl** sichern,
- Extraktion der Rohdaten aus dem ankommenden Stream in eine **VisionDomain** Datenstruktur,
- Verschiedene Filter anwenden und bereitstellen um die Datenstruktur zu bearbeiten,
- die von CONTROL angeforderten Visualisierungstechniken mittels verschiedenster Mapper darstellen und
- das Rendern des so erstellten Szenegraphen.

3.6.2 Kommunikation mit CONTROL über das NET-Modul

Das Netzmodul ist für die Kommunikation des VISION-Moduls mit dem CONTROL-Modul zuständig. Dazu werden die von CONTROL gesendeten Protokollbefehle in Java-Methodenaufrufe umgesetzt. Diese Methoden werden über den **Visionwrapper** (vgl. 3.6.3) zur Verfügung gestellt.

3.6.3 Die Schnittstelle zum Netz-Modul

Die Schnittstelle zum NET-Modul wird durch den **VisionWrapper** realisiert (vgl. Spezifikation der Netzwerkkomponenten), welcher Teil der **VisionControl**-Komponente ist. Der **VisionWrapper** stellt im Wesentlichen eine standardisierte Schnittstelle dar, die über die Implementierung des Interfaces **Wrappable** durch **VisionControl** mit dem gesamten Modul kommunizieren kann.

3.6.4 VisionControl

VisionControl stellt die Kontrollinstanz der VISION-Komponenten dar. Auf eine entsprechende Anfrage werden die unterstützten Visualisierungs-Elemente instantiiert, abgefragt, alle benötigten Parameter gesammelt und in Form einer Parameterliste zurückgegeben, die dann sämtliche Visualisierungs-Elemente mit ihren Abhängigkeiten sowie eine Beschreibung der GUI-Darstellung enthalten (vgl. dazu und zu den folgenden Beschreibungen auch Kapitel 3.4.7).
Diese Parameterliste wird im dynamischen GUI des CONTROL-Moduls dem Benutzer vorgelegt, der dort dann eine Visualisierungstechnik und mit den dazugehörigen Parametern konfigurieren kann. Die fertig konfigurierte Parameterliste wird an **VisionControl** zurückgegeben, damit hier die Visualisierungspipeline für die ausgewählte Visualisierung vorbereitet werden kann.

Sobald die Visualisierung begonnen wurde, werden die ankommenden Rohdaten in die Vision-Domain (vgl. Kap. 3.6.9) integriert, auf der dann die konfigurierte Pipeline arbeitet.
VisionControl kann auf Anfrage des CONTROL-Moduls (also durch den Benutzer über die GUI) die Verarbeitung der Daten in der Pipeline starten, pausieren und stoppen. Außerdem wird hier auch das Erstellen von Bildern einer Visualisierung sowie die Filmerzeugung aus mehreren Bildern kontrolliert.

3.6.5 Das VISIONGRID Paket

Das VISIONGRID-Paket dient zur Visualisierung eines Gebiets. Die zur Verfügung gestellten Methoden werden von dem GRID-Modul benutzt, um neben den drei Konstruktions-Ansichten auch eine projizierte Ansicht zur Verfügung zu stellen.

3.6.6 VisionViewer

Mittels des **VisionViewers** wird das berechnete Ergebnis der ausgewählten Visualisierung dargestellt. Diese Visualisierung kann dann mit Hilfe der Maus rotiert, skaliert und verschoben werden, bevor diese Einstellungen als Grundlage für die Erzeugung des endgültigen JPEG-Bildes benutzt werden. Die gemachten Einstellungen dienen auch bei den nächsten Visualisierungs-Schritten als Grundlage, falls dann auf eine Bildschirm-Anzeige verzichtet wird.
Das JPEG-Format wurde aus mehreren Gründen gewählt:

- Es genießt eine weite Verbreitung und wird nicht nur von professionellen Betriebssystemen im Allg. genauso unterstützt wie von allen Windows-Varianten, sondern auch von den meisten Open-Source-Betriebssystemen.

- Es ermöglicht gute Bildqualität bei geringem Platzverbrauch.

- Die Überführung einer Reihe von JPEG-Bildern in einen Film des Quicktime-Formates (vgl. Kapitel 2.5.2) ist vergleichsweise einfach.

3.6.7 VisionMovie

Mittels des **VisionMovie** Pakets kann aus den JPEG-Bildern der einzelnen Visualisierungs-Schritte ein Apple-Quicktime-Film generiert werden. Dazu werden alle erzeugten Bilder einer Visualisierung benutzt.
Das Quicktime-Format wurde aus folgenden Gründen gewählt:

- Es wird von verfügbaren Java-Bibliotheken unterstützt (JMF, vgl. Kapitel 2.6.1), was beim vielfach bevorzugten MPEG-Format nicht der Fall ist.

- Es ist nicht nur einfach abzuspielen, sondern auch mit vertretbarem Rechenaufwand zu generieren.

- Um einer befürchteten Plattformabhängigkeit wegen proprietärer Abspiel-Software vorzubeugen, konnte ein kleines Abspiel-Programm in Java realisiert werden.

3.6.8 Die Visualisierungs-Pipeline des Vision-Moduls

Der allgemeine Aufbau des Vision-Moduls orientiert sich stark am Aufbau bewährter Grafik-Systeme. Die Idee der Pipeline besteht darin, einen Datenfluß so zu bearbeiten, so daß aus den Rohdaten die für die ausgewählte Visualisierungstechnik irrelevanten Daten herausgefiltert, die relevanten Daten ausgewertet, und diese dann graphisch dargestellt werden.
Dabei können drei unterschiedliche Pipeline-Komponententypen unterschieden werden:

- Filter, die das große, eingehende Datenvolumen auf ein für die gewünschte Simulation-optimiertes Volumen reduzieren,

- Mapper, die die Zuordnung zwischen den gefilterten Daten und Java3D-Objekten übernehmen, und

- Renderer, die für das kompilieren der Java3D-Szene-Graphen und das Verwalten der Konfiguration des virtuellen Universums zuständig sind.

Die verschiedenen Module arbeiten dabei auf einer gemeinsamen Datenstruktur.

3.6.9 Die Datenstruktur

Die Datenstruktur für die Visualisierungspipeline setzt sich aus der Domain des Frameworks (**devisor.framework.foundation.domain**) und den GMV-Daten zusammen. Um den speziellen Anforderungen der verschiedenen Visualisierungstechniken zu entsprechen, wurden jeweils neue Klassen erstellt, welche von den Foundation-Klassen (**devisor.framework.foundation**) erben und diese um die benötigten Funktionen erweitern. Mittels dieser Klassen ist es möglich, über Nachbarschaftsbeziehungen innerhalb eines Gebietes die einzelnen Elemente anzusprechen, um auf diesen Berechnungen durchzuführen. Die GMV-Daten liefern neben den aktuellen Werten (wie Druck, Temperatur, etc...) zu den einzelnen Zeitschritten der Berechnung auch Informationen über die Verfeinerung der einzelnen Elemente. Um diese Informationen mit der Domain verschmelzen zu können, gibt es das **devisor.vision.ds** Paket.

Datenstruktur - VisionDomain

Die **VisionDomain** stellt die Datenstruktur des Grobgitters für die Visualisierungen dar. Die **VisionDomain** ist im wesentlichen für zwei Aufgaben zuständig: die Datenstruktur zu erstellen und sie dann in jedem Zeitschritt zu aktualisieren.
Die **VisionDomain** ist in der Lage, selbstständig aus einer **Domain** eine **VisionDomain** zu erstellen. Dazu müssen alle Objekte des **devisor.framework.foundation** Pakets in die entsprechenden Klassen des **devisor.vision.ds** Pakets umgewandelt werden. Wenn dies geschehen ist, müssen die Referenzen auf die Kinder bzw. Eltern dieser Objekte wieder aktualisiert werden.
Dazu werden die entsprechenden Vektoren der gegebenen **Domain** durchlaufen und in den Vektoren der **VisionDomain** gespeichert. Alle Objekte können in den Vektoren der **VisionDomain** über ihre eindeutige Nummer angesprochen werden. Das Suchen von benachbarten Elementen kann über die Referenzen auf die Kinder und Eltern der Objekte geschehen.

```
/* converting domain elements in vision elements */
       //nodes
       if (getNodes().size() != 0) {
           //System.out.println("creating VisionNodes");
           for (int i = 0; i < getNodes().size(); i++) {
               ValueNode newNode = new ValueNode(
                             (Node) getNodes().elementAt(i));
               valueNodes.add(newNode);
           }
       }
```

Codebeispiel 4: Erstellung einer VisionDomain

Danach müssen die Referenzen der neuen Objekte aktualisiert werden. Die Notwendigkeit hierfür besteht darin, das die ursprünglichen Klassen Referenzen auf Objekte der **devisor.framework.foundation** Klassen besaßen, aber Referenzen auf die **devisor.vision.ds** Klassen benötigt werden.

```
   /**
    * This method updates the childern of the edges. <br>
    * After that edges are tranformed to visionedges, their
    * children have to be set up properly which means that
    * the reference to the nodes has to be changed in a
    * reference to a corresponding visionnode
    */
   public void updateEdgesChildren() {
       // for all edges
       for (int i = 0; i < getEdges().size(); i++) {
           // getting the children of the edge
           int startNodeNumber =
             ((Node)((Edge)edges.elementAt(i)).
                  getStartNode()).getNumber();

           int endNodeNumber =
             ((Node)((Edge)edges.elementAt(i)).
                 getEndNode()).getNumber();

           // setting up children of the visionedge
           VisionEdge actualEdge = ((VisionEdge)visionEdges.
               elementAt(i));
           actualEdge.setStartNode(
               getValueNode(startNodeNumber) );
           actualEdge.setEndNode(
               getValueNode(endNodeNumber) );
       }
   }
```

Codebeispiel 5: Referenzaktualisierung

Um die Werte der GMV-Daten für jeden aktuellen Zeitschritt in den ValueNodes zu speichern wird der **GMVWrapper** benutzt. Nachdem alle Informationen aus den GMV-Daten mittels

des **GMVWrapper** gelesen wurden, werden diese unter Benutzung der folgenden Methode in die Domain integriert:

```
public void setValues(GMVWrapper aWrapper)
```

Zur Aktualisierung der Knotenwerte wird der Knotenvektor der **VisionDomain** durchlaufen, wobei an jedem Knoten

- Platz für die Werte gemacht wird und
- skalare und vektorielle Werte gespeichert werden.

In der **VisionDomain** sind die Knoten (ihre Koordinaten und skalaren bzw. vektoriellen Werte), sowie deren Elternobjekte enthalten. Zusätzlich sind in den GMV-Daten auch Knoten (Koordinaten und Werte) des Feingitters enthalten.

Um zu vermeiden, daßfür jeden Zeitschritt für jede Zelle des Grobgitters (also für jedes Element der **VisionDomain**) ein Feingitter erstellt wird, bietet die **VisionDomain** die Methode

```
public FineGrid createFineGridCell(int aGridCellNumber)
```

an. Die Motivation für diese Methode besteht darin, daß zu jedem Zeitschritt, je nach ausgewählter Visualisierungstechnik, nicht auf jeder Zelle des Grobgitters gearbeitet werden muß. Mit der genannten Methode kann in diesem Fall nur für benötigten Zellen ein Objekt der Klasse **FineGrid** erstellt werden, mit welchem man nun Berechnungen durchführen kann.
Damit die **VisionDomain** in der Lage ist, ein Feingitter-Objekt zu erstellen, müssen jedoch für jeden Zeitschritt die kompletten Informationen der GMV-Daten präsent sein. Der Gewinn besteht darin, daß zwar alle Daten gespeichert werden, der tatsächliche Rechenaufwand (Erstellen der Objekte, Ermittlung und Setzen der Referenzen) sich jedoch nur auf die wirklich angeforderten Zellen beschränkt.
Die GMV-Daten beinhalten zusätzlich folgende Informationen.

- Feingitter-Information:
  Diese Information besteht darin, daß für alle Elemente des Gebiets (z.B. Quads, Hexas, Tris und Tetras) beschrieben wird, welche Knoten sie enthalten.
  Der **GMVWrapper** liest diese Informationen und erstellt damit jeweils ein Objekt der Klasse **GMVCell**, welches er in einen entsprechenden Vektor speichert. Dieser Vektor wird beim Aktualisieren der **VisionDomain** übergeben.
- Feingitter Knoten (Koordinaten)
- Feingitter Werte (skalare, vektorielle)

Auf diese Informationen wird dann, wenn die Methode **createFineGridCell** benutzt wird, zugegriffen, und damit das gewünschte Objekt der Klasse **FineGrid** erstellt. Zugriff auf ein FineGrid-Objekt erfolgt über seine Nummer, die der Nummer der verfeinerten Grobgitterzelle entspricht.

Datenstruktur - FineGrid

Wie im Grundlagenkapitel bereits erwähnt, ermöglichen Zellen des Grobgitters immer (bis auf konstruierte Trivialbeispiele) nur Berechnungen ungenügender Präzision. Um genauere Berechnungen durchführen zu können, sind mehr Gitterpunkte nötig, was durch ein Feingitter (repräsentiert durch die Klasse **FineGrid**) erreicht werden kann. Wie im Abschnitt über die **VisionDomain** (Kapitel 3.6.9) bereits vorgestellt, bietet die **VisionDomain** alle Methoden, um für die benötigten Zellen ein Feingitter zu erstellen.
Das **FineGrid**-Objekt ist von ähnlicher Struktur wie die VisionDomain: Für jeden Elementtyp ist ein Vektor enthalten, der die Elemente, aus denen das Feingitter besteht, enthält. Die Nummer, über welche das Feingitter angesprochen werden kann, entspricht der Nummer der Grobgitterzelle, die es verfeinert.

Die wichtigste Methode der **FineGrid**-Klasse ist:

```
public void createElements()
```

Diese Methode übernimmt folgende, zur Erstellung des Feingitters notwendigen Aufgaben:

- Ermittlung der Anzahl der neuen Elemente.
  Die Anzahl der Elemente, welche pro Grobgitterzelle neu hinzukommen, ist abhängig vom Verfeinerungsfaktor, welcher durch den **GMVWrapper** ermittelt wird und von dem Elementtyp, aus welchem das Gebiet besteht (vgl. Kapitel Mehrgitter).

- Erstellung der neuen Elemente.
  Insbesondere müssen hierbei erst die Kinderelemente neu geschaffen werden und dann die entsprechenden Referenzen gesetzt werden. Hinzu kommt die wichtige Aufgabe, den neuen Elementen eindeutige Nummern zu geben, welche konform zu den Elementnummern der **VisionDomain** sowie der Nummerierung der GMV-Daten sind.

Die Elementnummern der neuen Elemente werden ebenso den GMV-Daten entnommen, wie die Koordinaten und Nummern der neuen Knoten. Da die GMV-Daten keine Kanten enthalten, müssen diese neu berechnet werden: Dabei wird auf die Spezifikation der Mehrgitterdaten zurückgegriffen, indem die Knoten nach der durch das Mehrgitterverfahren festgelegten Nummerierung zu Kanten zusammengefügt werden. Diese werden ebenso nach dem festgelegten Verfahren zu Elementen zusammengesetzt.
Die Nummer wird berechnet, indem ermittelt wird, wieviele Kanten sich vor dieser Kante befänden, wenn jede Gitterzelle verfeinert würde. Diese Anzahl muß zu der Gesamtanzahl aller Kanten der Domain addiert werden, um die Nummer der aktuellen Kante zur erhalten (vgl. dazu Codebeispiel 6).
Ebenso wie jedes einzelne Element der **VisionDomain** eine Interpolationsmethode besitzt, hat auch das Feingitter eine solche Methode. Sie basiert auf folgender Idee: Wenn ein Feingitter vorhanden ist, soll der Wert, der für diese Zelle ermittelt wird, möglichst aus nah beieinander liegenden Knoten interpoliert werden. Wenn eine Grobgitterzelle gefunden wurde, die einen gegebenen Punkt enthält, zu dem ein Wert interpoliert werden soll, dann unterscheidet sich das Ergebnis, das aus den Knoten der Grobgitterzelle berechnet wird, möglicherweise stark von dem Ergebnis, das aus den Knoten der Feingitterzelle berechnet wird.

Ein Testdurchlauf mit Debug-Ausgabe belegt diesen Unterschied:

```
neu: (-0.6315087895788593, 0.002490700615176221)
alt: (-0.24942118554156428, 0.31097942923820826)
neu: (-0.2484128520181635, 0.3074341664388298)
alt: (-0.2484128520181635, 0.3074341664388298)
neu: (-0.2570267201596842, 0.2939423851236014)
alt: (-0.2570267201596842, 0.2939423851236014)
neu: (-0.25917263573803867, 0.2657353241525613)
alt: (-0.25917263573803867, 0.2657353241525613)
```

Wobei “neu” der interpolierte Wert eines Partikels in einer Feingitterzelle beschreibt und “alt” der Wert desselben Partikels ermittelt durch eine Grobgitterzelle.

```
/**
 * This methods computes the next free number for a
 * new edge which can then be added to the
 * corresponding vector
 */
public int getNextFreeEdgeNumber() {
    int edgeNumber = 1;
    int type = myDomain.getType();
    // quantity of new elements per cell
    int quantityOfRefinedElements = 0;
    switch (type) {
    case VisionDomain.ELEMENT_TYPE_HEXA:
         quantityOfRefinedElements = 8;
        break;
    case VisionDomain.ELEMENT_TYPE_QUAD:
         quantityOfRefinedElements = 4;
        break;
    case VisionDomain.ELEMENT_TYPE_TRI:
         quantityOfRefinedElements = 3;
        break;
    case VisionDomain.ELEMENT_TYPE_TETRA:
         quantityOfRefinedElements = 4;
        break;
    }
    if (myDomain.getVisionEdges().size() != 0) {
        edgeNumber=((VisionEdge)myDomain.getVisionEdges()
            .lastElement()).getNumber();
        edgeNumber=edgeNumber+
           getNumber()*(int)Math.pow(
               quantityOfRefinedElements,
               refinementLevel  )+1;
    }
    if (visionEdges.size() != 0) {
        edgeNumber = ( (VisionEdge)
            visionEdges.lastElement() ).getNumber()+1;
    }
    return edgeNumber;

}
```

Codebeispiel 6: Ermittlung neuer Elementnummern

Auf den folgenden zwei Bildern sind die Wege von Partikel zu erkennen, die mittels eines Feingitters ermittelt wurden, und Wege, die ausschließlich mittels eines Grobgitters ermittelt wurden. Auch hier ist erkennbar, daß die Werte und dadurch auch die Wege der Partikel sich unterscheiden.

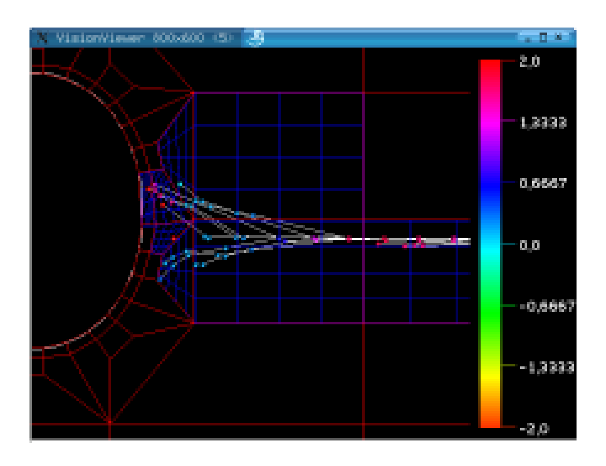


Abbildung 3.3: mit Feingitter

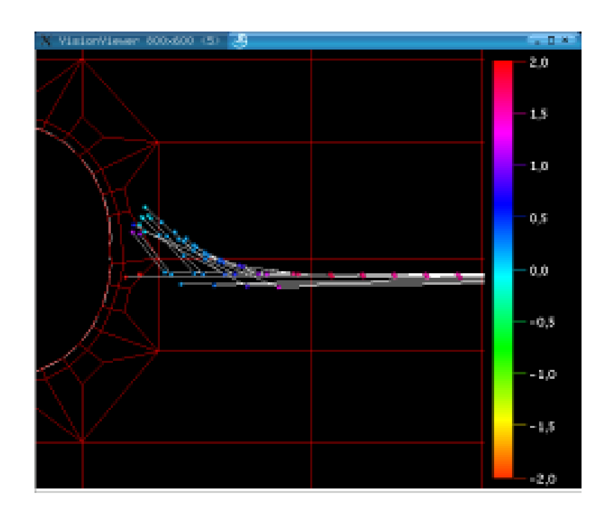


Abbildung 3.4: ohne Feingitter

3.6.10 Filter

Der Identitätsfilter (IdentityFilter)

Der Standard-Filter. Er schreibt alle Daten, die er erhält, unverändert in den Ausgabestrom. Sinnvoll läßt er sich durch Angabe eines Dateinamens zur Speicherung des ersten eingehenden Datenstroms verwenden.

Der Schwellwertfilter (ThresholdFilter)

Hier werden alle Werte über bzw. unter dem konfigurierten Schwellwert abgeschnitten. Bestehen die Daten aus einem Vektorfeld, wird der Schwellwert jeweils mit der euklidischen Norm der Vektoren verglichen.
Dadurch wird das Mapping auf interessante Bereiche der Simulation konzentriert, was sich ebenfalls in einem Geschwindigkeitsgewinn bemerkbar machen dürfte.

Der CutlinesFilter, CutplanesFilter

Cutlines und Cutsurfaces bilden die Ausnahme bei der Zuordnung von Visualisierungstechnik und Mapper. Hier besteht die Aufgabe darin, eine Ebene oder einen Raum zu durchschneiden und für die entstehende Schnittfläche/-linie eine neue Domain zu erzeugen und die Datenwerte für die entstehende Struktur zu interpolieren.
Somit erledigt dieser Filter die Hauptarbeit der Visualisierungstechnik. Die vielfältigen Anwendungsmöglichkeiten dieses Filters entstehen daraus, daß jeder beliebige Mapper mit diesem Filter benutzt werden kann. So könnte aus einer Schnittfläche zum Beispiel ein Shaded Plot berechnet werden.

Der Isolinienfilter

Dies ist ein Filter für die Visualisierungstechnik Isolinien (s. Kap. 2.4.3). Dieser Filter erstellt eine neue Domain in der Punkte mit gleichen Werte zu einem Linienzug verbunden werden. Die entsprechenden Isowerte zur Ermittlung der Isolinien können beliebig eingestellt werden.

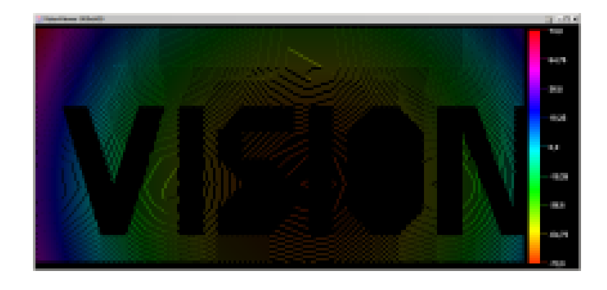


Abbildung 3.5: Beispiel für Isolinien

Der Surface3D Filter

Dieser Filter ermöglicht es, Knoten aus 2D-Daten entsprechend der Größe ihrer Werte in einer dreidimensionalen Form darzustellen. Es wird dazu eine neue Domain mit Z-Koordinaten erzeugt, die aus den skalaren Werten der entsprechenden Knoten und einem Skalierungsfaktor gewonnen werden. Der Skalierungsfaktor dient hierbei zur Verbesserung der Darstellung, um die entstehende Domain weder zu ”verzerrt” noch zu glatt darstellen zu können.

3.6.11 Mapper

Domain Mapper

Der **Domain Mapper** stellt eine Ausnahme unter den Visualisierungstechniken dar, da ihm, im strengen Sinne, keine wissenschaftliche Visualisierungstechnik zugrunde liegt. Er beschränkt sich auf die Darstellung der jeweiligen Domain und seiner Elemente, also im wesentlichen auf das Grobgitter. Dabei stellt er standardmäßig die Elemente mit bestimmten Farben dar:

- Tri-Elemente durch blaue Kanten,
- Quad-Elemente durch rote Kanten,

- Tetra-Elemente durch grüne Kanten und
- Quad-Elemente durch rote Kanten.

Des weiteren bietet der Mapper die Möglichkeit, folgende Boundary-Objekte der Domain darstellen zu lassen:

- Kugeln,
- Zylinder,
- Würfel,
- Dreiecke und
- 3D-Knoten

Die Farbauswahl und die Wahl des Aussehen der Boundary-Objekte sind über die Parameterlisten einstellbar.

Glyph Mapper

Es handelt sich um eine ganze Gruppe von Mappern, die Vektorfelder durch Symbole visualisiert (s. Kap. 2.4.4). Mit diesen Symbolen oder auch Glyphs wird das Objekt näher beschrieben. Es gibt verschiedene Arten von Glyphs, die die Bewegung des Objektes wiedergeben. Glyphs können durch Vektoren in Form von Pfeilen oder durch Punktmengen in Form von Kugeln dargestellt werden. Natürlich gibt es eine Reihe anderer Gestalten, die unter Glyphs aufgefasst werden können. In dieser PG hat man sich jedoch auf Pfeile und Kugeln beschränkt. Dabei werden diese Pfeile bei einem zwei dimensionalen Objekt flach dargestellt, während die Kugeln entsprechend als Kreise erkennbar werden. Ein Pfeil kann viele Aufgaben erledigen bzw. kann viele Informationen gleichzeitig dem Betrachter veranschaulichen. Er hat eine Länge, einen Radius, eine Farbe, und eine Ausrichtung. Es werden alle Informationen ausgenutzt, jedoch werden sie der Übersichtlichkeit halber nicht gleichzeitig beobachtbar sein. Bei jeder Einstellung hat der Benutzer die Möglichkeit, jeweils eine Visualisierungsart auswählen. Da die Ausrichtung implizit mitenthalten ist, hat der Benutzer bei jedem Pfeil zwei Informationen. Konkreter ausgedrückt, hat man bei jeder Visualisierung eine Information über die Ausrichtung und die Farbe, über die Ausrichtung und die Länge oder über die Ausrichtung und die Dicke des Pfeils. Wird eine dieser drei Angaben (Farbe, Dicke oder Länge) gewählt, so sind die anderen Eigenschaften konstant, d.h. sie nehmen bestimmte konstante Werte an. Dabei haben diese drei Eigenschaften keine vordefinierte Bedeutung, sie können vom Benutzer für jede mögliche Visualisierungsart jeweils benutzt werden. Die Pfeile können benutzt werden, um zum Beispiel die Geschwindigkeit darzustellen. Die Ausrichtung des in Bewegung schreitenden Objektes wird durch die Richtung des Glyphs an einem Punkt erkennbar. Die Geschwindigkeit selbst wird nach Einstellung durch die Farbe, der Länge oder durch den Radius bestimmt.
Außer der Geschwindigkeit sind mit diesen Glyphs noch andere Simulationswerte visualisierbar. Manchmal ist jedoch die implizite Richtungsangabe nicht erwünscht bzw. irrelevant, so daß man auf die kugelförmigen Glyphs wechselt. Eine Kugel kann jedoch maximal zwei Informationen - die Farbe und der Radius - enthalten. Es kann wieder der Übersichtlichkeit halber jeweils nur zwischen einer der beiden Informationen gewählt werden. Kugeln oder Kreise (im

2D) werden z.B. für den Druck oder für die Temperatur verwendet. Dabei bleibt die nicht gewählte Eigenschaft konstant.

Wie werden die Glyphs angeordnet?

Da das Objekt aus vielen Knoten besteht, hat man an jedem Knoten eine oder mehrere Informationen, die den Glyph aufspalten. Hierfür ist es nicht angemessen, alle Knoten mit Glyphs zu gestalten, da das Objekt in den Hintergrund gerät bzw. nicht erkennbar wird. Außerdem wird der Verlauf der Glyphs durch ihre hohe Anzahl nicht mehr klar erkennbar, da sie sich unter Umständen verzweigen bzw. überlagern. In diesem Fall hat der Benutzer die Möglichkeit die Anzahl der Glyphs einzugeben, wobei die max. Angabe die Anzahl der Knoten des Grobgitters ist.

Auf diese Weise waren vier Mapper nötig, um das gewünschte Ergebnis zu realisieren.

- Arrow3DMapper : Für die Darstellung der dreidimensionalen Pfeile
- Arrow2DMapper : Für die Darstellung der zweidimensionalen flachen Pfeile
- Sphere3DMapper : Für die Darstellung der Kugeln
- Sphere2DMapper : Für die Darstellung der Kreise

Shaded Plot Mapper

Dies ist der Mapper für die Visualisierungstechnik Shaded Plot (s. Kap. 2.4.3). Er interpoliert für jeden darzustellenden Punkt zwischen den in der **VisionDomain** enthaltenen Knoten den dazugehörigen Wert und ordnet ihm eine Farbe zu. Es entsteht (je nach Konfiguration) eine Art Farbüberlauf über die Flächen oder Kanten, der den Werteverlauf über der Domain widerspiegelt.
Um dies zu bewerkstelligen, müssen die 2D-Daten der Domaindatenstruktur so aufbereitet werden, daß diese in den Szenegraphen eingehängt werden können. Dabei wird zunächst zwischen **VisionEdges**, **VisionTris** und **VisionQuads** unterschieden. Dies ist nötig, weil die Kanten der Polygone für die spätere Visualisierung in der richtigen Reihenfolge erstellt werden müssen.
Abhängig von den skalaren Werten, die danach auf einem bestimmten Knoten berechnet wurden, wird diesem ein bestimmter Farbwert zugeteilt. Die Zuordnung dieses Farbwertes erfolgt über das Legendenmodul des CONTROL-Pakets.



Abbildung 3.6: Beispiel für einen Shaded Plot

Particle Tracing Mapper

Dieser Mapper stellt die Möglichkeit zur Verfügung, die Bewegung virtueller Teilchen (Partikel) in einem Vektorfeld zu berechnen und darzustellen. (s. auch S. 44)
Das Particle Tracing basiert auf folgendem Algorithmus (vgl. dazu Kap. 2.4.4):

1. Ermittlung der aktuellen Position $p(t_i)$ des Teilchens zum Zeitpunkt t_i.

2. Interpolation zwischen Gitterpunkten um den Geschwindigkeitsvektor an der Stelle $p(t_i)$ zu bekommen.

3. Integration der Gleichung

$$p'(t) = V(p(t), t)$$

   um $p(t_{i+1})$ zu bekommen, wobei das Vektorfeld $V(p(t), t)$ einen Vektor in Abhängigkeit eines Punktes $\in \mathbb{R}^3$ und des Zeitschritts t liefert.

Zellsuche
Um den Wert eines Partikels ermitteln zu können, muß das Element gefunden werden, in welchem sich das Partikel befindet. Jedes Element der **VisionDomain** (**VisionHexa**, **VisionTetra**, **VisionQuad**, **VisionTri**) bietet daher die Methode

```
public boolean isInElement(Point2d aPoint2d)
```

bzw.

```
public boolean isInElement(Point3d aPoint3d)
```

an, anhand welcher getestet werden kann, ob das Element den gegebenen Punkt enthält (vgl. entsprechende Klassen für den Testalgorithmus). Um die Geschwindigkeit einer Suche zusätzlich zu erhöhen, werden die Methoden

```
public boolean isInBoundingBox(Point2d aPoint2d)
```

und

```
public boolean isInBoundingBox(Point3d aPoint3d)
```

angeboten, welche den Test auf die achsenparallele Boundingbox des Elements reduziert.
Beide Suchtypen werden in der Klasse **ParticleTracingMapper** realisiert.

Zellsuche - globale Suche
Zum Start jeder Berechnung wird für jedes Partikel eine lineare Suche über alle Elemente durchgeführt. Hierbei wird der entsprechende Vektor durchlaufen und seine Elemente mit der Boundingbox-Methode überprüft. Nur wenn dieser Test positiv ausfällt, wird die aufwendigere Methode **isInElement** ausgeführt. Sollte ein Partikel genau auf einer Kante, bzw. auf einer Seitenfläche eines Elements liegen, wird das Element zurückgeliefert, welches als erstes in dem Vektor überprüft wird. Der Interpolations-Algorithmus (vgl. entsprechende **devisor.vision.ds** Klassen) liefert bei allen Elementen in diesem Fall das gleiche Ergebnis.

Zellsuche - Nachbarschaftsbeziehung
Wurden bereits eine Berechnung zur Ermittlung von Partikelbewegungen durchgeführt, so sind bereits Informationen über deren Positionen vorhanden.
Diese Informationen werden in der Klasse **ParticleFunction** gespeichert.

Beide Suchtypen werden in der Klasse **ParticleTracingMapper** durch die Methode

```
public VisionHexa getNextHexa(
    Point3d point,
    VisionHexa oldHexa)
  throws HexaNotFoundException
```

realisiert (für jedes Element gibt eine entsprechende Methode).
Das übergebene Element ist das Element, in welchem sich das Partikel zu dem letzten Zeitschritt befand. Entspricht es dem *null*-Objekt, so wird eine lineare Suche über alle im entsprechenden Vektor enthaltenen Elemente gestartet:

```
/* hexa == null
 * wich means looking for the first hexa of
 * a computation
 */
} else {
    Vector hexaVector=myDomain.getVisionHexas();
    Enumeration e=hexaVector.elements();
    while(e.hasMoreElements() && !(hexaFound)) {

        aktHexa=(VisionHexa)e.nextElement();

        if(aktHexa.isInBoundingBox(point)) {
            if(aktHexa.isInElement(point)) {
                hexaFound=true;
                return aktHexa;
            }
        }
    }
}
```

Codebeispiel 7: Lineare Zellsuche

Man beachte, daß hier erst der Boundingboxtest angewandt wurde, um die Effizienz zu steigern.
Ist beim Aufruf der Suchmethode ein Element vorhanden, in dem sich das Partikel zum letzten Zeitschritt befand, so wird die Suche über die Nachbarschaftsbeziehung gestartet. Das Prinzip dieser Breitensuche basiert auf den Eltern-Kind-Beziehungen der **devisor.vision.ds**-Objekte. Jedes Objekt hat das Element als Parent, von welchem es ein Teil darstellt. So hat ein **VisionQuad** beispielsweise vier **VisionEdges** als Kinder. Diese Kinder haben umgekehrt das **VisionQuad** als Parent eingetragen.
In einem ersten Schritt werden alle benachbarten Elemente ermittelt und in einen Vektor eingetragen. Das Element, in dem sich das aktuelle Partikel vorher befand, wird ebenfalls in diesen

Vektor eingefügt (und zwar als erstes Element). Danach werden diese Elemente mittels der bereits vorgestellten Methoden untersucht, ob sie das aktuelle Partikel enthalten. Ist das der Fall, dann wird das entsprechende Element zurückgeliefert, ansonsten wird das Verfahren auf die benachbarten Elemente ausgeweitet.

```
        /* hexa != null
         * which means beeing at one (inner) step
         * of a computation
         */
        if(aktHexa!=null) {
            neighbours.addElement(aktHexa);

            while(size>0 &&
                  !(hexaFound) &&
                  neighbourAvailable)
            {
                neighbourAvailable=false;

                if( (aktHexa=(VisionHexa)neighbours.remove(0))
                     != null)
                {
                    if(aktHexa.isInBoundingBox(point)) {

                        if(aktHexa.isInElement(point)) {
                            hexaFound=true;
                            return aktHexa;
                        }
                    }
                    int allNeighbours=aktHexa.
                        getNeighbourHexas().length;

                    for(int i=0;i<allNeighbours;i++)
                        if(aktHexa.getNeighbourHexas()[i]
                                                     !=null)
                            neighbourAvailable=true;d

                    for(int i=0;i<allNeighbours;i++)
                        if(aktHexa.getNeighbourHexas()[i]
                                                     !=null)
                            neighbours.addElement(aktHexa.
                                getNeighbourHexas()[i] );
                }
                size--;
            }
```

Codebeispiel 8: Zellsuche über Nachbarschaftbeziehung

Die Nachbarelemente werden durch die Methode

```
public VisionHexa[] getNeighbourHexas()
```

folgendermaßen ermittelt:

- Ermittlung der Kinderelemente des Elements, in welchem sich das Partikel vorher befand.
- Ermittlung der Eltern der eben ermittelten Kinderelemente.
- Einfügen aller Eltern, unter Vermeidung von Dubletten, in einen Vektor.

Dieser Vektor enthält dann alle Nachbarelemente. Die Suche endet, wenn alle Elemente der **VisionDomain** durchsucht wurden. In diesem Fall hat das Partikel entweder die Domain verlassen oder es wurde außerhalb der Domain platziert.

Informationsspeicherung
In der bereits oben erwähnten Klasse **ParticleFunction** wird zwar die Lage jedes Partikels gehalten. Damit ist aber nicht aber die genaue Position gemeint, sondern nur das Element, in dem sich das Partikel befindet. Da es für jedes Partikel eine Partikelfunktion gibt, kann die Suche über Nachbarschaftsbeziehungen bereits mit dieser Information realisiert werden (vgl. die Klasse **ParticleTracingMapper**).
Die wichtigste Methode dieser Klasse ist:

```
public void eval (double t, double[] actualPoint)
```

Diese Methode liefert zu einem als **double-**Array übergegebenen Punkt den zugehörigen Wert zurück. Da die Funktion ausschließlich für das Particle Tracing benutzt wird, werden nur die vektoriellen Werte zurückgeliefert.
Die Auswertung beinhaltet im wesentlichen die Interpolation innerhalb des Elements, welches das Partikel enthält (vgl. **getInterpolatedVector**-Methoden der entsprechenden **devisor-.vision.ds**-Klassen).

Integration der Differentialgleichung
Die Integration der Differentialgleichung ist nötig, um die neue Position eines Partikel zu ermitteln. Hierfür wird das Paket **devisor.vision.particle** benutzt.
Nachdem der Wert eines Partikels ermittelt wurde, kann nun mit Hilfe der Klasse **RungeKutta** die oben genannte Gleichung gelöst werden. Das heißt, es wird die aktuelle Position und der Wert an dieser Position benötigt. Die aktuelle Position wird wie oben erwähnt in der Klasse **ParticleFunction** gespeichert. Der aktuelle Wert wird ermittelt, indem die vom aktuellen Element angebotete Methode **getInterpolatedVector** benutzt wird.
Sind diese Werte vorhanden, wird die Methode

```
public void step()
```

benutzt, um das Verfahren von Runge-Kutta vierter Ordnung anzuwenden, wodurch die neue Position ermittelt wird. An die Koordinaten der neuen Position kann dann über

```
public final double [] getY ()
```

zugegriffen werden.
Um eine Berechnung starten zu können, muß die Methode

```
public final void setStart (double startX, double [] startY)
```

aufgerufen werden, durch welche der Startzeitpunkt gesetzt wird, sowie die Koordinaten des Partikels als Array.

Der Algorithmus realisiert durch den Mapper
Der Mapper kann durch einen inneren Zähler feststellen, ob er sich am Anfang einer Berechnung befindet, oder ob schon Informationen über Partikel vorliegen. Zu jedem Zeitschritt wird durch die **PipeInstance** die Methode

```
protected BranchGroup buildSceneGraph()
```

aufgerufen, so daß sich hier der Platz für die Erhöhung des Zählers befindet.
Die folgende Methode beinhaltet somit das Grundgerüst der Partikelverfolgung:

```
 protected BranchGroup buildSceneGraph() {
    this.checkFirstPoints();
    legend = new LegendLabel();
    legend.setColorsStyle(legendStyle, max, max * (-1));
    if (timestepCounter == 0) {
       // init
       initComputation();
       timestepCounter++;
    } else {
       // step
       oneStepOfComputation();
       timestepCounter++;
    }
    // rest
    myLogger.debug("Building scene graph...");
    BranchGroup bg = new BranchGroup();
    bg.addChild(mapEdges());
    if (showBoundaryGrid || showBoundaryShape) {
        bg.addChild(mapBoundaries());
    }
    bg.addChild(mapParticles());
    if (this.showFineGrid)
       bg.addChild(mapFineGridEdges());
    myLogger.debug("Finished building scene graph !");
    return bg;
}
```

Codebeispiel 9: Grundgerüst der Partikelverfolgung

Ist eine Berechnung am Anfang, d.h. der Mapper wird das erste Mal mit Daten gefüllt, werden folgende Schritte realisiert:

- Es wird das Element gesucht, in welchem sich das aktuelle Partikel befindet.
  Hierfür wird auf den **devisor.vision.ds**-Klassen eine lineare Suche durchgeführt.
  Ist ein entsprechendes Element vorhanden, wird das Partikel als **Point3D** oder **Point2D** in den Vektor

  ```
  private Vector firstPoints;
  ```

aufgenommen.
Zusätzlich wird die Anzahl der Partikel, welche in einer Berechnung berücksichtigt werden soll, in **particlesAliveCount** gespeichert. Für jedes Partikel wird dann auch noch sein Zustand (d.h. ob es noch innerhalb des Gebiets ist oder nicht) in

```
particleIsAlive = new boolean[particlesAliveCount];
```

gespeichert.

- Es wird für jedes Partikel ein Objekt der Klasse **ParticleFunction** und eines der Klasse **RungeKutta** erstellt.

Im Gegensatz dazu werden während einer Berechnung für jedes "lebendiges" Partikel folgenden Schritte ausgeführt:

- Die neue Position des Partikel wird berechnet,
- es wird überprüft, ob ein Element existiert, welches diese neue Position enthält. Ist dies der Fall, wird es in dem Vektor

  ```
  private Vector particlesTraces
  ```

  gespeichert. Dieser Vektor besteht aus Vektoren, welche jeweils für ein Partikel alle Positionen beinhalten.

Die oben genannten Punkte werden jeweils nach den Elementtypen und nach der Dimension der Punkte unterschieden und ausgeführt.

Die Visualisierung
Die Visualisierung wird mittels einer **BranchGroup** realisiert. Das Hinzufügen von Kindern und die dadurch enstehenden Linien und Partikeln werden in der Methode

```
protected BranchGroup mapParticles()
```

erschaffen. Hier werden je nach Visualisierung die Partikel mit Linien verbunden.

3.6.12 Renderer

Der Renderer verwaltet die Einstellungen des Java3D-Universums und initialisiert den Kompilierungsvorgang auf dem vom Mapper bereitgestellten Szene-Graphen.
Dazu werden zwar ausschließlich Klassen der Java3D-API benutzt, aber dennoch erschien die Kapselung in eine selbständige Klasse und einen separaten Teil der Pipeline sinnvoll, um spätere Modifikationen so weit wie möglich zu vereinfachen.

Kapitel 4

Projektdurchführung und Organisation

4.1 Zeitlicher Ablauf

4.1.1 Seminarphase

Die Arbeit der Projektgruppe begann mit einer Seminarphase, in der jeder Teilnehmer über Grundlagen des Projektgruppenthemas referierte. Folgende Themen wurden behandelt:

- Grundlagen der 3D-Grafik,
- Wissenschaftliche Visualisierung,
- Java 3D,
- Netzwerkprogrammierung mit Java,
- Java Swing,
- Together und CVS,
- Einführung in die existierende Simulationssoftware,
- Visualisierungstool AVS (Beispiel für ein existierendes modulares Visualisierungssystem),
- computational steering,
- Datenrepräsentation und -Formate und
- Grundlagen aus der numerischen Mathematik.

Viele der behandelten Themen finden sich auch in diesem Endbericht wieder (s. Kap. 2).

4.1.2 Spezifikationsphase

Zu Beginn des Wintersemesters stand zunächst eine kleine Orientierungsphase auf dem Programm. Die Teilnehmer machten sich mit dem Zyklus der numerischen Strömungssimulation und der Bedienung von FEATFLOW exemplarisch am „Membranproblem" vertraut.
Nach etwa drei Wochen begann die eigentliche Spezifikation. Zuerst wurden im großen Plenum Ideen gesammelt, und es wurde sich auf Minimalziele geeinigt. Schon früh stand das modulare Konzept, so daß die Einteilung in Kleingruppen vorgenommen wurde. Die detaillierte Spezifikation wurde dann in diesen Kleingruppen vorgenommen, wobei allerdings in regelmäßigen Abständen Zwischenergebnisse im Plenum diskutiert wurden und die Spezifikation der Modulkonnektivität vorgenommen wurde. Parallel zur Planung wurden auch bereits erste Prototypen definiert.
Während der Semesterferien fanden keine Aktivitäten statt.

4.1.3 Implementierungsphase

Zeitgleich mit Beginn des Sommersemesters wurde mit der Implementierung begonnen. Schon bald stellte sich heraus, daß die Spezifikation in weiten Teilen nicht vollständig war. Insbesondere mußten die Visualisierungspipeline und das Netzwerkprotokoll deutlich überarbeitet werden.
Nach Ende der sehr zeitaufwendigen Implementierung wurden die Module integriert und getestet. Die Integration ging – sicher wegen der im zweiten Anlauf dann gut durchgeführten Spezifikation – recht problemlos voran.
Das offizielle Ende der Projektgruppenarbeit stellt dieser Endbericht dar.

4.2 Einteilung in Kleingruppen

Während der Planungsphase wurde die folgende Einteilung in Kleingruppen beschlossen.

Name	Modul
Hendrik Becker	VISION
Christian Engels	VISION
Markus Glatter	VISION
Dominik Göddeke	NET
Eduard Heinle	NET
Mathias Kowalzik	CONTROL
Patrick Otto	VISION
Wissam Ousseili	VISION
Thomas Rohkämper	GRID
Mathias Schwenke	GRID
Nicole Skaradzinski	CONTROL
Tom Vollerthun	VISION

Gegen Ende der Implementierungsphase wurde diese Einteilung naturgemäß immer weiter gelockert.

Anhang A

Benutzerhandbuch

A.1 Benötigte Pakete

Zunächst wird Java 2 SDK, ab Version 1.4.x benötigt, um DEVISOR ausführen zu können.

Für VISION wird Java3D benötigt und zwar ab der Version 1.3.1. Auch das aktuellste Java-Media-Paket JMF sollte installiert sein.

Die neuesten `gcc`- und `g77`-Compiler (ab Version 3.2) werden für FEATFLOWbenötigt.

Die aktuellste Distribution von FEATFLOW ist für das korrekte Arbeiten mit dem DEVISOR-Paket unumgänglich, genauso, wie die als Teil von NEXUSdistribuierten FEATFLOW-Patches.

A.2 Installation

Am Anfang müssen folgende Verzeichnisse erstellt werden:

```
/devisor/
/devisor/solverengines/
/devisor/devisor/
```

A.2.1 Installation des Numerikpakets FEATFLOW

Die folgenden Schritte beschreiben Schritt für Schritt die Vorgehensweise bei der FEATFLOW-Installation:

1. Die aktuelle Distribution von FEATFLOW kann von `www.featflow.de` heruntergeladen werden. Alternativ kopiert man sich die Datei von der PG-CD (`pgcd.tar.gz`).
2. Dann entpackt man alles in `devisor/solverengines`.
3. Man wechselt in das Verzeichnis `devisor/devisor/server/featflow_mod`.

4. Man rufe das Programm `install_patches` auf, womit auch implizit `install_featflow` gestartet wird, wobei vorher die unten definierten Umgebungs-Variablen gesetzt werden müssen.

5. Bei der Konfiguration wählt man die Architektur aus, auf der FEATFLOWlaufen soll, auf Intel-Rechnern wäre das also die Option `b: Intel/x86 Linux (EGCS)`.

6. Performance libraries sollen nicht benutzt werden.

7. AVS Modules sollen auch nicht benutzt werden.

8. Auf die Frage "Start installation?" gebe man y ein.

9. Nach dem Kompilieren sollten alle weiteren Fragen mit "no" beantwortet werden..

A.2.2 Installation von DEVISOR

Zum Installieren einfach das entsprechenden Installationsprogramm für die benutzte Plattform ausführen und den Anweisungen folgen. Es werden alle Module inklusive des NEXUSservers installiert, allerdings muß dieser noch für das Zielsystem übersetzt werden.

A.2.3 Installation des Servers

Der Server ist zusammen mit DEVISOR installiert worden. Um ihn zu übersetzen und startbereit zu machen, gehe man in das unten definierte $DEVISOR_SERVER-Verzeichnis und gebe `make` ein. Nun wird der Server übersetzt. Die Datei `modules.conf` muß an den entsprechenden Computer angepasst werden, um die vorhandenen Module zur Verfügung zu stellen. Anschließend wird der Server mit `./nexus` gestartet.

A.2.4 Einrichtung der DEVISOR-Umgebung

Um DEVISOR starten zu können, müssen einige Umgebungsvariablen gesetzt sein:

DEVISOR Das Verzeichnis, in das DEVISOR(das Verzeichnis `/devisor/`) installiert wurde

DEVISOR_SERVER Das Verzeichnis, in dem sich der Server befindet, z.B. `/devisor/devisor/server/`

DEVISOR_SERVERPPORT Ein beliebiger Port, an dem der Server lauschen soll, z.B. 4711

DEVISOR_SOLVERENGINES Das Verzeichnis, in das FEATFLOW installiert wurde, z.B. `devisor/solverengines`

DEVISOR_WORKSPACE Irgend ein Arbeitsverzeichnis, z.B. `/devisor/devisor/-server/workspace`

FEATFLOW Das Verzeichnis von FEATFLOW, z.B. `/devisor/solverengines/featflow1.2/featflow`

ARCH die Architektur, auf der FEATFLOW benutzt wird, `ultra2` oder `intelp2`

A.3 Schnellstart: Simulation des Membranproblems

An dieser Stelle wird exemplarisch die Vorgehensweise bei der Simulation und Visualisierung des Membranproblems auf einem lokal laufenden NEXUS-Server mit dem DEVISOR-Modul Schritt für Schritt und modulübergreifend beschrieben.

Als Voraussetzung wird angenommen, daß der Server NEXUS und das DEVISOR-Paket installiert und kompiliert sind (siehe Kap. A.2).

Nun folgt eine Übersicht über das Vorgehen in diesem Kapitel. Diese Schritte repräsentieren die übliche Vorgehensweise bei der Bedienung des Systems.

1. Starten des Servers NEXUS und CONTROL.
2. Konfiguration des Servers und ein neues Projekt anlegen.
3. Starten von NUMERICS und Berechnung von ein paar Schritten des Moduls **FeatPoisson**
4. Starten von VISION und Visualisierung der Rechenschritten mit dem **ShadedPlot**-Filter

Der Server NEXUS wird mit dem Aufruf von `./nexus` von dem Verzeichnis `$DEVISOR_SERVER` gestartet. Nun wechselt man in das DEVISOR-Verzeichnis und gibt `ant run_control` ein.

Damit wird das Modul CONTROL gestartet. Es erscheint das Hauptfenster. Nun muß der Server konfiguriert werden. Man gehe unter das Menü **Server** und dort zum Menüpunkt **Serververwaltung**. Dort ist schon der Server ”Alias” auf dem `localhost` und dem Port 4711 voreingestellt. Den wählt man aus. Falls man mit den Einstellungen zufrieden ist, so klickt man auf den **Aktualisieren**-Button. Man sieht, falls die Aktualisierung gutgeht, in dem Modulbeschreibungs-Fenster die von dem Server angebotenen Module. Mit dem **Schließen**-Button wird dieser Dialog geschlossen.

Als nächstes wird ein neues Projekt erstellt. Dies geht über das Menü **Projekt**, **Neues Projekt**. Dort wählt man ein Verzeichnis in einem Standard-Datei-Auswahl-Dialog aus, in dem man das neue Projekt anlegen möchte. Man kann einfach `tmp_devisor` eingeben und mit **OK** bestätigen, dann wird das neue Verzeichnis in dem Startverzeichnis des aktuellen Benutzers angelegt. Ein neues Benutzerfenster wird eingeblendet, bei dem oben die verfügbaren Module angezeigt werden.

Im folgenden wird das Szenario geschildert um auf einem fertigen Gitter eine Simulation zu starten. Dazu klicke man auf **NUMERICS** in dem neu erschienenen Fenster. Es wird ein neues TabPane mit dem Namen **NUMERICS** erstellt. Dort muß man nun den Server auswählen, das ist der vorher augewählte **Alias**-Server. Neben dem Server wählt man **FeatPoisson** als das Module aus. Nun wird auch der Startknopf freigegeben, auf den man dann klicken muß. Das Module wird gestartet und meldet sich mit einem **`OK Module Started`** in dem unteren Textfenster. Jetzt muß das Problem konfiguriert werden. Dies geht mit dem nun freigeschalteten Button **Typ**. Dieser öffnet das Konfigurations-Fenster. Nun soll man ein Problem auswählen.

Bei dem Modul FeatPoisson steht nur das Membranproblem zur Auswahl.

Wenn man nun auf **Weiter** klickt, werden die Parameter, die zur Konfiguration des Membranproblems notwendig sind, eingeblendet. Man kann hier wieder auf **Weiter** klicken, denn eine Gitterdatei wurde hier schon per Default angegeben, oder man wählt eine andere Gitterdatei im FEAST-Format unter dem **GridParameter**, **Domain laden**. Nun kommt die Modul-Konfiguration. Diese ist auch schon von vorne hinein richtig eingestellt, so daß man auf den Button **Ende** klicken kann.

Nun ist man wieder beim Ausgangs-Fenster angelangt. Hier kann man auf den **Start**-Button klicken, um die Simulation des Membranproblems zu starten. Ein paar Schritte kann man das System rechnen lassen, bevor man mit VISION die Visualisierung startet. Den Vorgang kann man sehr gut in dem unteren Status-Fenster beobachten.

Beim Start von VISION werden die Schritte bis zu der Problemkonfiguration analog ausgeführt. Es muß ein Filter, ein Mapper sowie ein Renderer ausgewählt werden. Es empfiehlt sich den **Identity-Filter**, den **ShadedPlot-Mapper** und den **Standard-Renderer** auszuwählen. Die Komponenten der Pipe sind bereits für dieses Problem passend konfiguriert, so daß ein zusätzliche Konfiguration optional ist. Nachdem man auf **Weiter** geklickt hat, muß man die Domain, auf welcher die Pipe arbeitet auswählen. Man beachte, daß das die selbe Domain sein muß, die auch das Numeric-Modul zur Berechnung der Daten benutzt hat. Als letztes muß nur noch das Numeric-Modul ausgewählt werden.
Wenn alles eingestellt ist, dann kann man auch VISION starten. Es erscheint ein VISION-typisches Fenster, in dem die Visualisierung der Domain anhand der Auslenkung der Membran dargestellt wird.

Um eine ideale Visualisierung zu erstellen, kann nun der Blickwinkel und die Entfernung des Gebiets in dem sich öffnenden Fenster eingestellt werden. Nach dieser Justierung kann der **Ok**-Knopf geklickt werden. Für jeden Zeitschritt wird nun ein *offline-Bild* erstellt und ein QuickTime-Movie von allen Zeitschritten zusammen erstellt.
Alle anderen Module sind analog aufgebaut, so daß die Bedienung des Programms nach dem Durchspielen des gerade vorgestellten Beispiels sehr einfach ist. Die Bedienung unterscheidet sich hauptsächlich in den Problem- und Parametereinstellungen für die Module. Für die nähere und detailliertere Beschreibung der Bedienung der einzelnen Module wird auf die nachfolgenden Handbücher verwiesen.

A.4 CONTROL – Die Steuerung des Systems

A.4.1 Einleitung

Mit CONTROL ist es möglich, komplette Simulationen durchzuführen, indem über CONTROL alle für eine Simulation benötigten Module zentral gesteuert und konfiguriert werden können. Dazu werden alle für eine Simulation benötigten Module in Projekten verwaltet. Die Module können über eine dynamische GUI konfiguriert und über eine Kassettenrekorderfunktion gesteuert werden. Es kann jederzeit die Verbindung zu einem Modul abgebrochen und zu einem späteren Zeitpunkt wieder aufgenommen werden. Zusätzlich ist es möglich, bei den Modu-

len vorliegende Ergebnissse zu downloaden und, falls dies vom Modul unterstützt wird, sich Statistiken über den Verlauf anzeigen zu lassen.

A.4.2 Installation

Zum Installieren einfach das entsprechenden Installationsprogramm für die benutzte Plattform ausführen und den Anweisungen folgen. Es werden alle Module inklusive des NEXUSservers installiert, allerdings muß dieser noch für das Zielsystem übersetzt werden (siehe Kapitel A.2, S.108).

Voraussetzungen

Als Voraussetzungen wird mindestens Java 2 SDK Version 1.4.1 benötigt und eine Java 3D SDK Version 1.3 erwartet.

Linux und Unix-Systeme

Nach Abspeichern des Installationsprogramms eine Shell öffnen, in das Verzeichnis des Installationsprogramms wechseln und mit **#sh ./install.bin** die Installation ausführen.

Windows-Systeme

Nach dem Download die Installation durch Doppelklick auf **install.exe** starten.

A.4.3 Allgemeine Beschreibung

CONTROL besteht aus einem Hauptfenster, in dem in internen Fenstern die Projekte verwaltet werden. Eine Steuerung der Projektverwaltung und Serververwaltung erfolgt über die Menüleiste des Hauptfensters, die Verwaltung der Module innerhalb eines Projekts hingegen erfolgt immer über die Menüleiste des internen Projektfensters.

A.4.4 Serververwaltung

Die Serververwaltung erfolgt unabhängig von der Projektverwaltung, wird jedoch in Projekten benutzt, um neue Server und Module auswählen zu können.

Neuen Server anlegen

Über ”`Server>Neuer Server`” kann ein neuer Server angelegt werden. Es erscheint der in Abb. A.2 gezeigte Dialog. Für den neuen Server müssen Alias, Host, Port und Benutzername angegeben werden, wobei der Alias eindeutig sein muß. Mit ”`Verbindung testen`” kann getestet werden, ob Verbindung zu dem Server aufgenommen werden kann.

Serververwaltung

Über ”`Server>Serververwaltung`” kann die Serververwaltung geöffnet werden. Es erscheint der in Abb. A.3 gezeigte Dialog. In der oberen Liste ”`Serverliste`” werden alle verwalteten Server angezeigt, das Textfeld ”`Serverbeschreibung`” enthält jeweils die

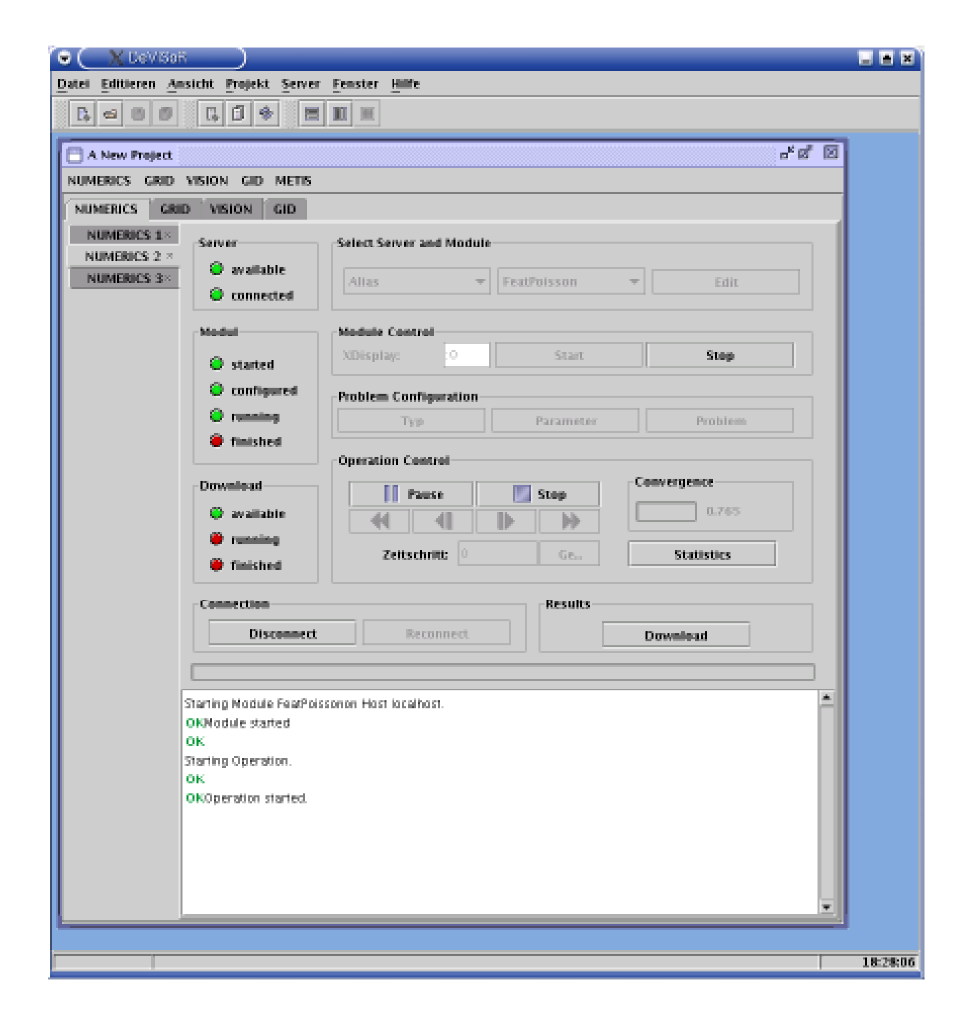


Abbildung A.1: Das Hauptfenster

Beschreibung (z.B. Betriebssystem) des ausgewählten Servers. In der unteren Liste ”`Modulbeschreibung`” werden alle Module angezeigt, die auf dem ausgewählten Server verfügbar sind. Unter ”`Ausgewähltes Modul`” kann man noch einmal die komplette Beschreibung des ausgewählten Moduls nachlesen. Es folgt nun eine Erläuterung der Buttons:

”`Hinzufügen`” Es erscheint der in Abb. A.2 gezeigte Dialog und es kann ein neuer Server wie oben beschrieben angelegt werden.

”`Editieren`” Es erscheint wieder der Dialog aus Abb. A.2, diesmal jedoch mit den Angaben des ausgewählten Servers.

”`Löschen`” Der in der Serverliste ausgewählte Server wird gelöscht.

”`Aktualisieren`” Die Liste der verfügbaren Module des ausgewählten Servers wird aktualisiert.

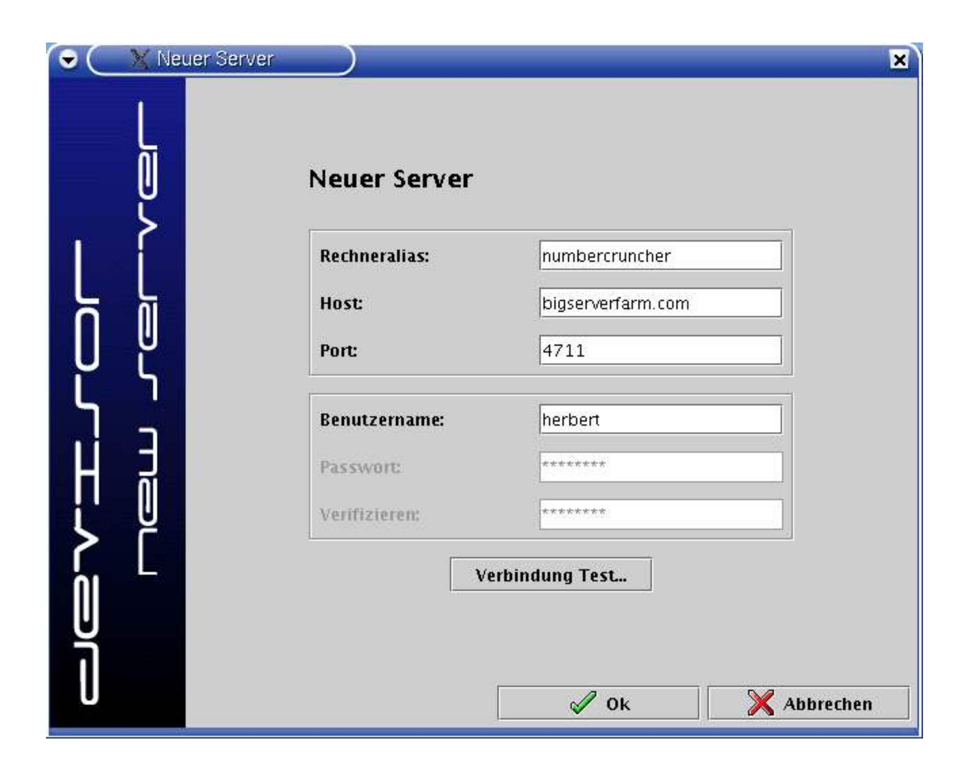


Abbildung A.2: Neuen Server anlegen

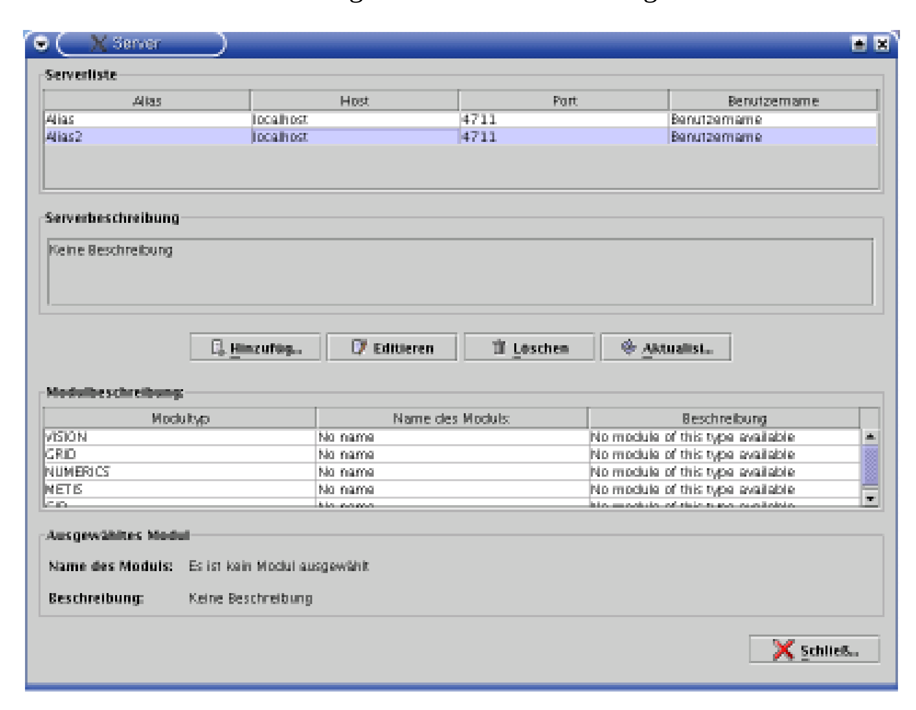


Abbildung A.3: Serververwaltung

Serverliste aktualisieren

Über ”`Server>Server aktualisieren`” wird die komplette Serverliste, d.h. die Modullisten aller verwalteter Server, aktualisiert.

A.4.5 Projektverwaltung

Es können Projekte neu angelegt, geladen, gespeichert und geschlossen werden.

Neues Projekt anlegen

Über ”`Datei>Neues Projekt`” oder ”`Projekt>Neues Projekt`” kann ein neues Projekt angelegt werden. Es erscheint der in Abb. A.4 gezeigte Dialog. In dem Dialog kann der

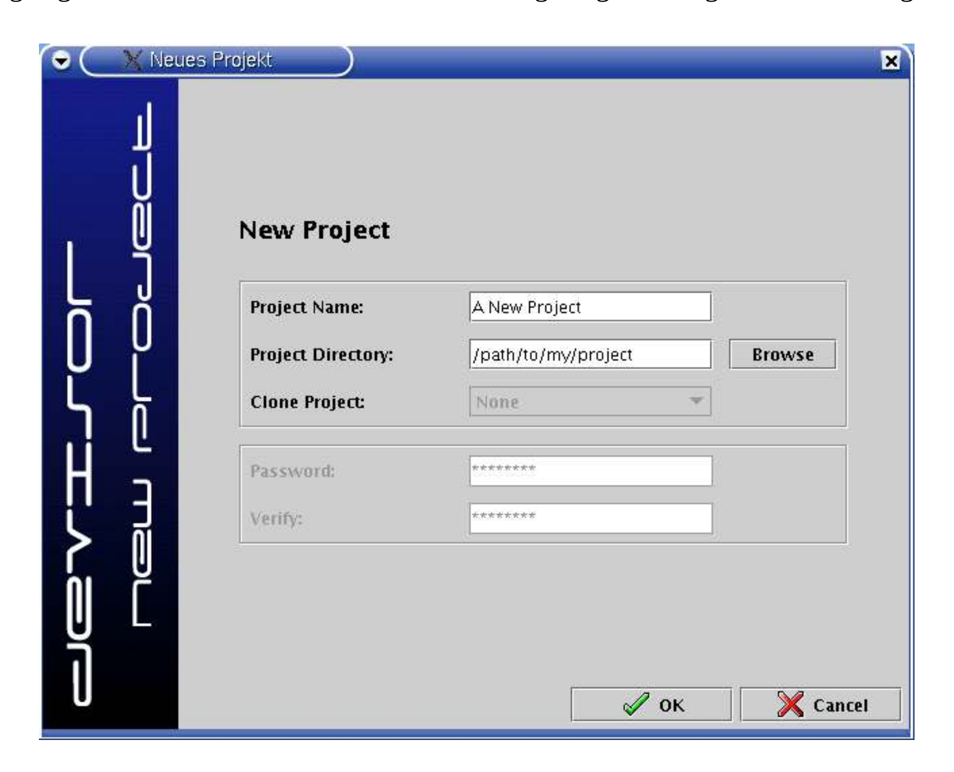


Abbildung A.4: Neues Projekt anlegen

Projektname und das Projektverzeichnis angegeben werden. Nach erfolgreicher Beendigung des Dialogs (”`OK`”-Button drücken), erscheint im Hauptfenster ein neues Projektfenster und es wird ein neues Projektverzeichnis angelegt.

Projekt laden

Über ”`Datei>Öffne Projekt...`” kann ein bereits existierendes Projekt geladen werden. Dazu erscheint ein Standarddialog zur Dateiauswahl. Gespeicherte Projekte haben immer die Endung `*.devisor`.

Projekt speichern

Über ”`Datei>Speichern`” wird das Projekt des aktiven Projektfensters gespeichert.

Projekt schließen

Über ”`Datei>Schließe Projekt`” kann das aktive Projektfenster geschlossen werden. Über ”`Datei>Alle schließen`” können alle geöffneten Projekte geschlossen werden.

A.4.6 Steuerung eines Moduls

In Abb A.5 ist ein Projektfenster zu sehen. Da alle Module gleich behandelt werden, besitzen auch alle Module die gleiche Oberfläche. Wieviel davon zur Verfügung steht, hängt vom jeweiligen Modul ab, z.B. kann es sein, daß keine Statistiken angeboten werden (z.B. beim GRID-Modul) oder die Kassettenrekorderfunktion nicht voll unterstützt wird. In den linken Panels

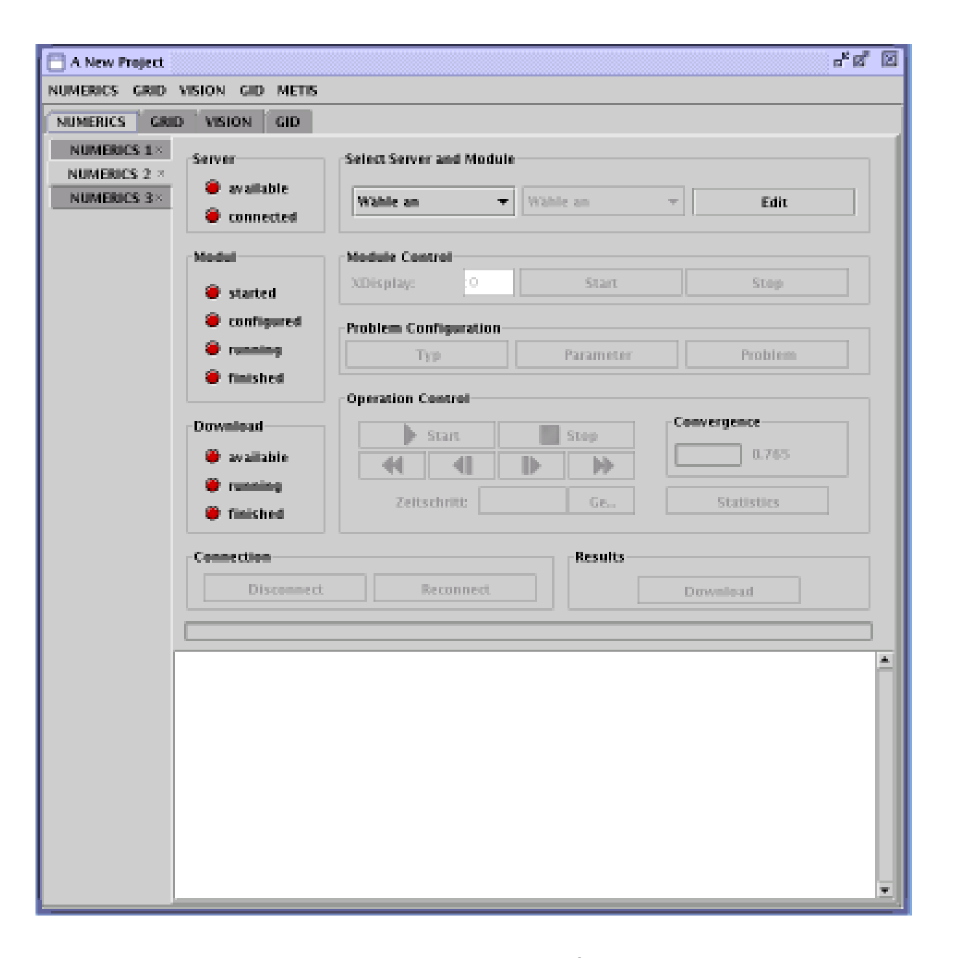


Abbildung A.5: Projektfenster

”`Server`”, ”`Modul`” und ”`Download`” wird durch rote und grüne LEDs der Status eines Moduls angezeigt. Über die Modulleiste des Projektfensters können Module neu geöffnet, neu angebunden oder geschlossen werden. Die Modulpunkte sind nach Modultypen sortiert. In den horizontalen TabbedPanes sind die Module nach Modultypen sortiert, Module eines Modultyps hingegen sind in vertikalen TabbedPanes angeordnet.

Neues Modul

Über "`NUMERICS>Neues NUMERICS`" kann z.B. ein neues Numerikmodul erstellt werden. Analog können Module anderer Modultypen erstellt werden.

An laufende/inaktive Module anbinden

Über "`NUMERICS>Öffne NUMERICS`" kann z.B. ein bereits laufendes/inaktives Modul neu angebunden werden. Dazu erscheint der in Abb. A.6 gezeigte Dialog. Analoges gilt für Module anderer Modultypen.

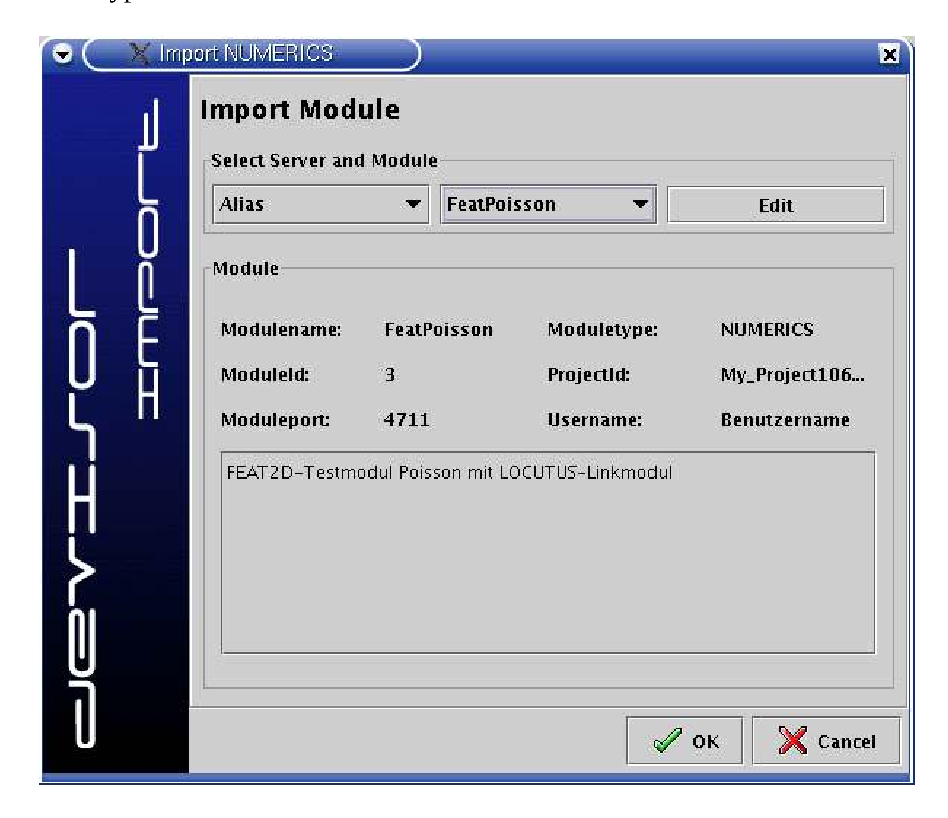


Abbildung A.6: Module importieren

Modul schließen

Über "`NUMERICS>Schließe NUMERICS`" kann z.B. das aktive Modul eines Modultyps geschlossen werden. Das gleiche gilt für alle anderen Modultypen. Über "`NUMERICS>Alle schließen`" können alle Module eines Modultyps geschlossen werden.

Modul auswählen

Die Modulauswahl erfolgt in dem Panel "`Select Module and Server`". Falls noch kein Modul ausgewählt wurde, gibt es zwei Möglichkeiten zur Modulauswahl: zum einen kann man ein Modul über die zwei Comboboxen wählen. In der ersten Combobox werden die Aliase aller

Server aus der Serverliste angezeigt. Nach Wahl eines Servers werden alle auf diesem Server verfügbaren Module in der zweiten Combobox angezeigt. Nachdem eins dieser Module gewählt wurde, kann die Konfiguration beginnen. Die zweite Möglichkeit, ein Modul zu wählen, besteht darin, die Serververwaltung zu benutzen. Über den Button ”`Edit`” wird die in Abb. A.3 gezeigte Serverwaltung geöffnet. Der einzige Unterschied ist, daß statt eines ”`Schließen`”-Buttons ein ”`OK`”- und ein ”`Abbrechen`”-Button eingeblendet werden. Das gewählte Modul kann so mit ”`OK`” bestätigt werden. Sobald ein Modul gestartet wurde, kann es nicht mehr geändert werden, bis es gestoppt wird.

Modul starten/stoppen

Über die Buttons ”`Start`” und ”`Stop`” in dem ”`Module Control`”-Panel kann ein ausgewähltes Modul gestartet und wieder gestoppt werden. Beim Stoppen eines Moduls wird dieses auf der Serverseite beendet. In dem Feld ”`XDisplay`” kann angegeben werden, auf welchem Bildschirm das Modul laufen soll.

Modul konfigurieren

Die Modulkonfiguration erfolgt in dem Panel ”`Problem Configuration`”. Die Konfiguration eines Moduls ist nur möglich, wenn ein Modul in der Modulauswahl ausgewählt worden ist und dieses sich entweder in einem gestarteten, aber nicht laufenden Zustand oder im Pause-Zustand befindet. Dann erfolgt die Konfiguration des Moduls über die dynamische GUI. Falls ein Modul zum ersten Mal konfiguriert wird, muß über den Button ”`Gesamt`” das Modul einmal komplett konfiguriert werden. Nach erstmaliger Konfiguration kann zusätzlich der ”`Problem`”Button verwendet werden, um das ausgewählte Problem zu konfigurieren, ohne dieses neu auswählen zu müssen. Eine genauere Beschreibung der dynamischen GUI erfolgt in Kap. A.4.7.

Der Kassettenrekorder

Nachdem ein Modul gestartet und mindestens einmal konfiguriert wurde, steht die Kassettenrekorderfunktion in dem ”`Operation Control`”-Panel zur Verfügung. Hierbei wird der ”`Start`”Button von jedem Modul unterstützt, um das Modul zum Laufen zu bringen, alle anderen Funktionen sind hingegen optional und werden vom Modul festgelegt. Hier eine Auflistung der Funktionen:

”`Start`”	Das Modul läuft nun auf dem Server. Falls ”`Pause`” unterstützt wird, wechstelt ”`Start`” in ”`Pause`”, ansonsten wird der Button deaktiviert. Bei einem VISION- oder GRID-Modul z.B. wird bei ”`Start`” das Modul gestartet, bei einem Numerikmodul startet die Berechnung.
”`Stop`”	Bei einem Numerikmodul beispielsweise wird eine laufende Berechnung gestoppt.
”`Forward`”	Falls dies vom Modul unterstützt wird, kann z.B. in einem Numerikmodul ein Zeitschritt weiter gegangen werden.
”`Rewind`”	Falls dies vom Modul unterstützt wird, kann z.B. beim Numerikmodul ein Zeitschritt zurück gegangen werden.
”`Pause`”	Falls dies unterstützt wird, pausiert z.B. bei einem Numerikmodul die Berechnung und das Modul kann rekonfiguriert werden.
”`Gehe`”	Falls dies unterstützt wird, kann z.B. für ein Numerikmodul angegeben werden, bis zu welchem Zeitschritt gerechnet werden soll.

Verbindung

Über die Buttons ”`Connect`” und ”`Disconnect`” kann nun jeder Zeit die Verbindung zu einem gestarteten Modul abgebrochen und zu einem späteren Zeitpunkt wieder aufgenommen werden.

Statistik

Über den Button ”`Statistik`” kann zu jeder Zeit das Statistikmodul aktiviert werden, falls das jeweilige Modul die Wiedergabe von Statistiken unterstützt (üblicherweise Numerikmodule) und bereits statistische Daten zur Darstellung zur Verfügung stehen. Man gelangt über den Button zunächst in das Konfigurationsfenster des Statistikmoduls. Im angezeigten Dialog können beliebig viele Statistikdiagramme aus den ankommenden Daten generiert werden, es können so auch mehrere Datengruppen in einem Diagramm dargestellt werden, z.B. für Vergleichszwecke. Im oberen Teil des Konfigurationsdialoges kann durch die Buttons ”`Hinzufügen`” und ”`Entfernen`” jeweils ein weiteres Diagramm hinzugefügt bzw. das aktuell markierte entfernt werden. Die Buttons ”`Nach oben`” und ”`Nach unten`” verschieben das aktuell markierte Diagramm um jeweils eine Position nach oben oder unten, um die Möglichkeit zu schaffen, die gewünschten Diagramme individuell anzuordnen.
Im der Diagrammübersichtstabelle ist der ”`Diagrammname`” frei wählbar, das Feld ”`Ausgewählte Daten`” gibt die aktuelle Konfiguration eines Diagramms in Kurzform wieder. Das Feld ”`Diagrammtyp`” kann per Auswahlliste folgende Werte erhalten, die folgerichtig den Typ des Diagramms bzw. dessen Orientierung darstellen:

Balkendiagramm, horizontal erzeugt ein Balkendiagramm, das horizontal orientiert ist. Der Ausdruck ” horizontal” bezieht sich hierbei auf die Lage der Beschreibungsachse, so daß die Balken des Diagramms vertikal dargestellt werden.

Balkendiagramm, vertikal erzeugt ein vertikal ausgerichtetes Balkendiagramm.

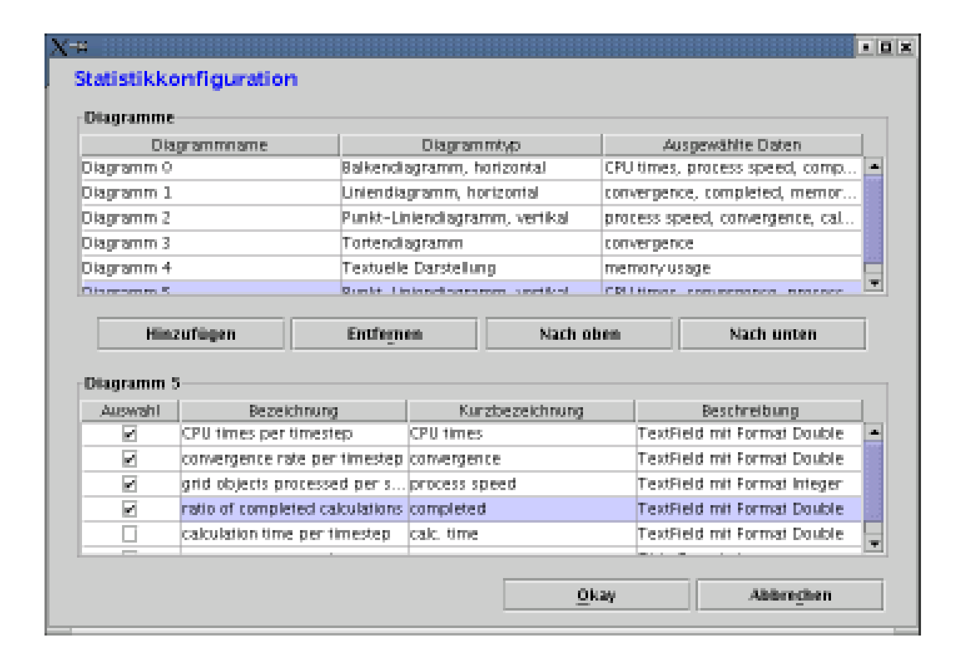


Abbildung A.7: Das Konfigurationsfenster des Statistikmoduls

Liniendiagramm, horizontal erzeugt ein horizontal ausgerichtetes Liniendiagramm.

Liniendiagramm, vertikal erzeugt ein vertikal ausgerichtetes Liniendiagramm.

Punktdiagramm, horizontal erzeugt ein horizontal ausgerichtetes Punktdiagramm.

Punktdiagramm, vertikal erzeugt ein vertikal ausgerichtetes Punktdiagramm.

Punkt-Liniendiagramm, horizontal erzeugt ein horizontal ausgerichtetes Liniendiagramm, dessen einzelne Wertepunkte zwecks Hervorhebung als Punkte dargestellt werden.

Punkt-Liniendiagramm, vertikal erzeugt ein ebensolches Diagramm in vertikaler Ausrichtung.

Tortendiagramm erzeugt ein Tortendiagramm aus den ankommenden Daten, zum Beispiel, um Verhältnisse darstellen zu können.

Textuelle Darstellung erzeugt kein Diagramm, sondern eine Textdarstellung der ankommenden Daten, um diese beispielsweise von Hand überprüfen zu können.

Wenn ein Diagramm im oberen Teil des Dialogfensters markiert ist, wird dessen aktuelle Konfiguration im unteren Teil angezeigt und kann bearbeitet werden. Für jede zur Verfügung stehende Datengruppe wird hier die Bezeichnung, Kurzbezeichnung und eine kurze Beschreibung ausgegeben. Wenn eine Datengruppe im aktuell zu konfigurierenden Diagramm angezeigt werden soll, kann diese Datengruppe durch einen Klick auf ”`Auswahl`” zum Diagramm hinzugefügt werden und ebenso wieder entfernt werden.
Durch einen Klick auf ”`Abbrechen`” wird der Vorgang der Konfiguration unterbrochen, ein Klick auf ”`Okay`” führt zur Erstellung der konfigurierten Diagramme und deren Darstellung in einem neuen Fenster. In diesem Fenster werden alle Diagramme angezeigt, und bei Eingang

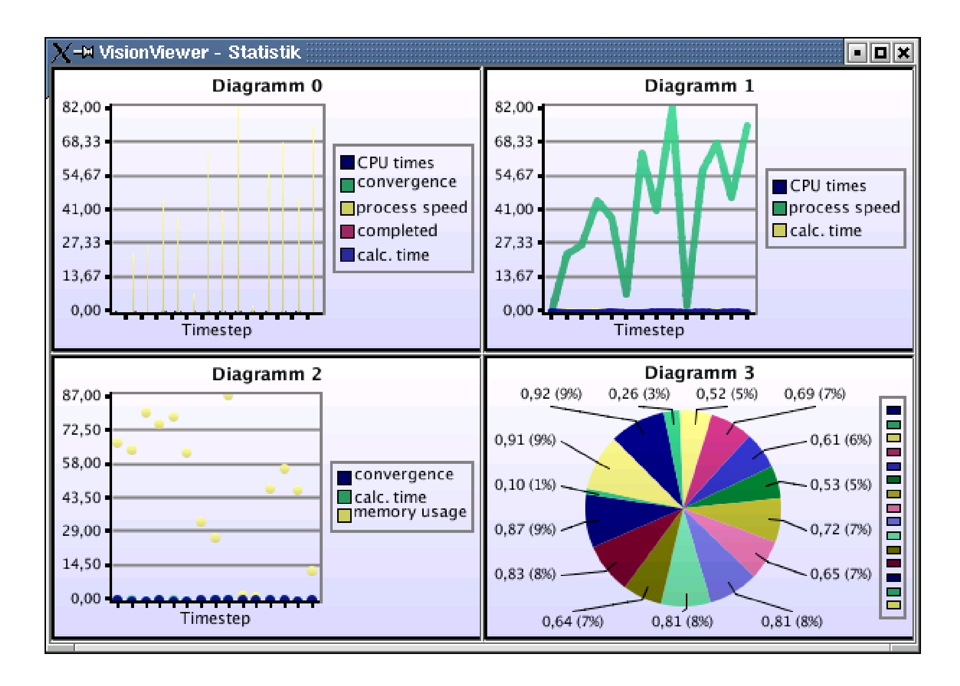


Abbildung A.8: Das Diagrammfenster

neuer Daten (über die Schnittstelle zu CONTROL) werden diese im Diagramm ebenfalls angezeigt. Über das Popup-menü dieses Fensters, welches auch hier durch Betätigen der rechten Maustaste ausgelöst werden kann, kann das Diagrammfenster als JPEG-Grafik abgespeichert werden. Nach Klick auf ”`Speichere Statistik als Bild...`” wird dieser in eine

Speichere Statistik als Bild...

Abbildung A.9: Das Popup-Menü

anzugebene Datei geschrieben, um so das Diagramm auch nach Abschluß der Berechnungen weiterhin zur Verfügung zu haben.

Download

Der Ergebnisdownload (s. Abb. A.10) erfolgt über den Downloaddialog. Hier werden in einer Tabelle die vorhandenen zum Download zur Verfügung gestellten Ergebnisse angezeigt, wobei zwischen einer vereinfachten und einer ausführlichen Auflistung der Eigenschaften des Ergebnisses gewählt werden kann. Bei beiden Anzeigemodi wird als Tooltip eine Detailübersicht des Ergebnisses angezeigt. Durch Doppelklicken kann bei zeitschrittabhängigen Ergebnissen der Zeitschrittauswahldialog geöffnet werden. Über das Kontextmenü oder über Downloadbutton wird ein Ergebnis in die Warteschlange der Downloads eingereiht; ebenfalls durch das Kontextmenü oder durch den Abbrechenbutton kann es natürlich wieder daraus entfernt werden. Über die Anzahl der Verbindungen legt man fest, wieviele Ergebnisse gleichzeitig heruntergeladen werden dürfen.

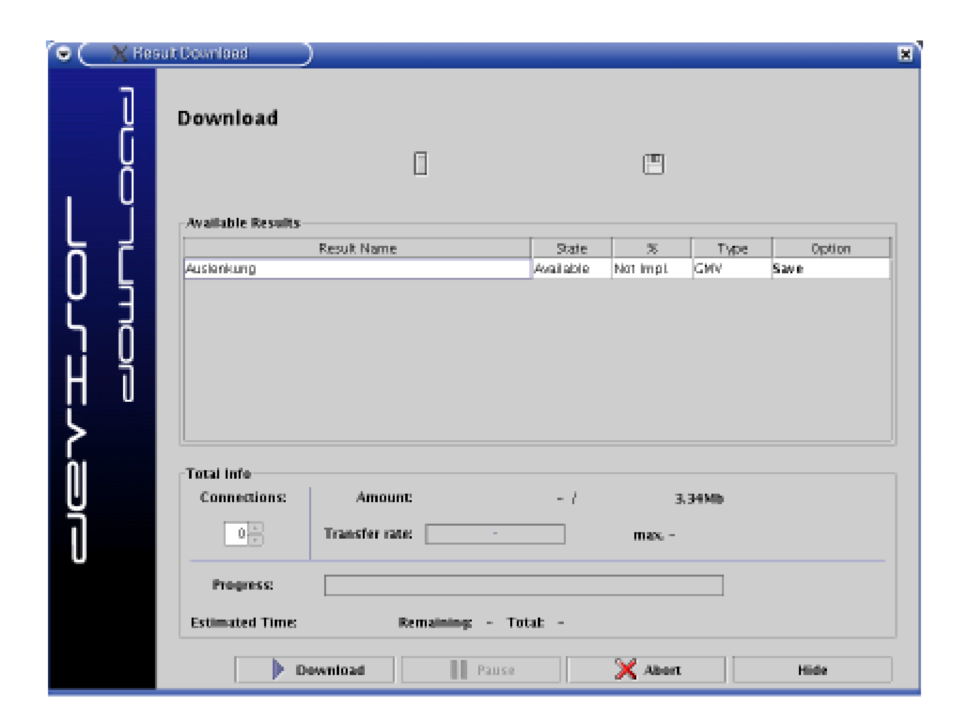


Abbildung A.10: Der Ergebnisdownload

A.4.7 Konfiguration eines Moduls

Für die Konfiguration eines Moduls steht die dynamische GUI zur Verfügung. Da die Konfiguration eines Moduls sich ändern kann und bei jedem Modul verschieden ist, werden hier die einzelnen Komponenten der dynamischen GUI unabhängig von irgendeinem bekannten Modul vorgestellt und gezeigt, wie man sie bedienen kann. Da in der dynamischen GUI immer Beschreibungen für die einzelnen Komponenten mitgeliefert werden, wird hier für die sinngemäße Benutzung der dynamischen GUI eines Moduls auf diese Beschreibungen oder auf die Handbücher des jeweiligen Moduls verwiesen.

Übersicht

Die Durchführung einer vollständigen Konfiguration eines Moduls besteht immer aus drei Schritten:

Schritt 1 Aus einer Liste von Problemen (z.B. Membranproblem) wird ein Problem ausgewählt.

Schritt 2 Das in Schritt 1 ausgewählte Problem wird konfiguriert.

Schritt 3 Das Modul wird unabhängig vom ausgewählten Problem konfiguriert.

Erst nach Beendigung des dritten Schritts ist die vollständige Konfiguration abgeschlossen.

Schritt 1 – Auswahl eines Problems

In Schritt 1 wird ein Problem aus einer Liste von Problemen ausgewählt. In Abb A.11 wird als Beispiel das Membranproblem gezeigt. Im unteren "`Ausgewähltes Problem`"-Panel ist die vollständige Beschreibung des ausgewählten Problems sichtbar. Über den "`Weiter`"-Button gelangt man zu Schritt 2. Falls das Modul zu einem früheren Zeitpunkt schon einmal konfiguriert wurde und bei dieser Konfiguration wieder das gleiche Problem gewählt wird, erfolgt eine Abfrage, ob die alten Einstellungen beibehalten werden sollen. Falls dies der Fall ist, sind die Einstellungen in Schritt 2 und Schritt 3 die gleichen wie beim letzten Mal. Ansonsten werden in Schritt 2 und Schritt 3 die Default-Einstellungen benutzt. Das gleiche gilt, falls man von Schritt 2 aus über den "`Zurück`"-Button zurück zu Schritt 1 geht und noch einmal das gleiche Problem wählt. Auf der linken Seite des Dialogs wird immer angezeigt, bei welchem

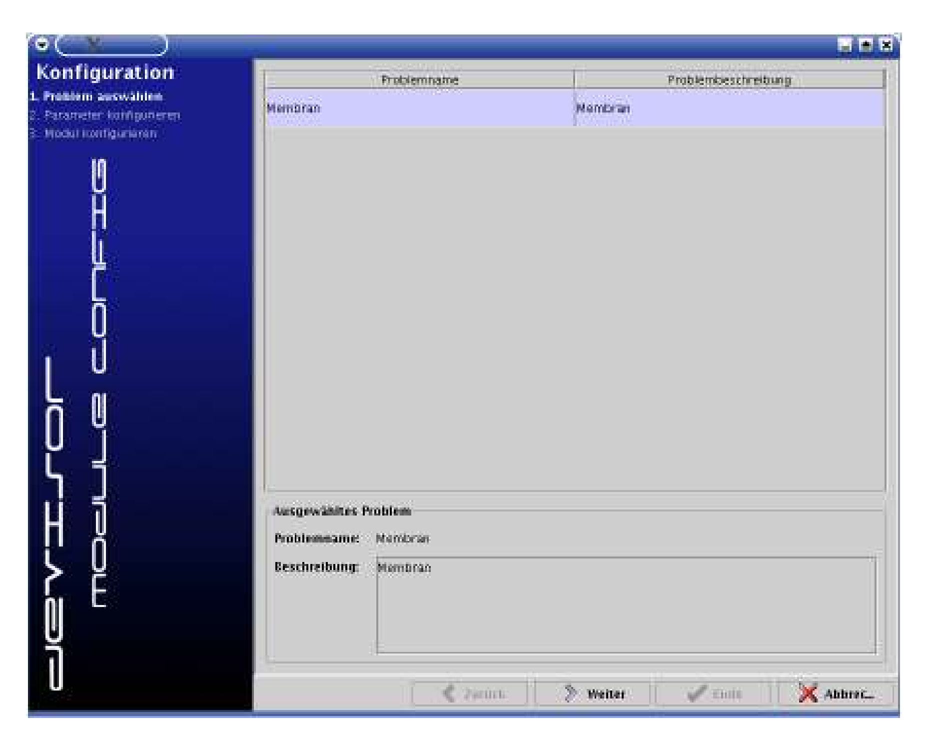


Abbildung A.11: Der Ergebnisdownload

Konfigurationsschritt man sich befindet.

Schritt 2 – Konfiguration eines Problems

In Schritt 2 wird nun das ausgewählte Problem konfiguriert. Da die dahinterliegende Struktur für eine mögliche GUI in Schritt 2 und Schritt 3 die gleiche ist, wird hier die GUI von Schritt 3 implizit mitbeschrieben. In Abb. A.12 ist ein Beispiel für eine mögliche GUI zu sehen. Auf der linken Seite ist ein Baum zu sehen. Jeder Knoten und jedes Blatt dieses Baums kann konfiguriert werden. Wenn ein Knoten oder Blatt ausgewählt wird, erscheint auf der rechten Seite das dazugehörige Panel. Dieses Panel ist i.d.R. aus mehreren einzelnen Panels aufgebaut, die untereinander angeordnet sind. Jedes dieser Panels stellt einen Parameter dar, der konfiguriert werden kann. Das Panel eines Knotens wird immer aus diesen Parameterpanels modular zu-

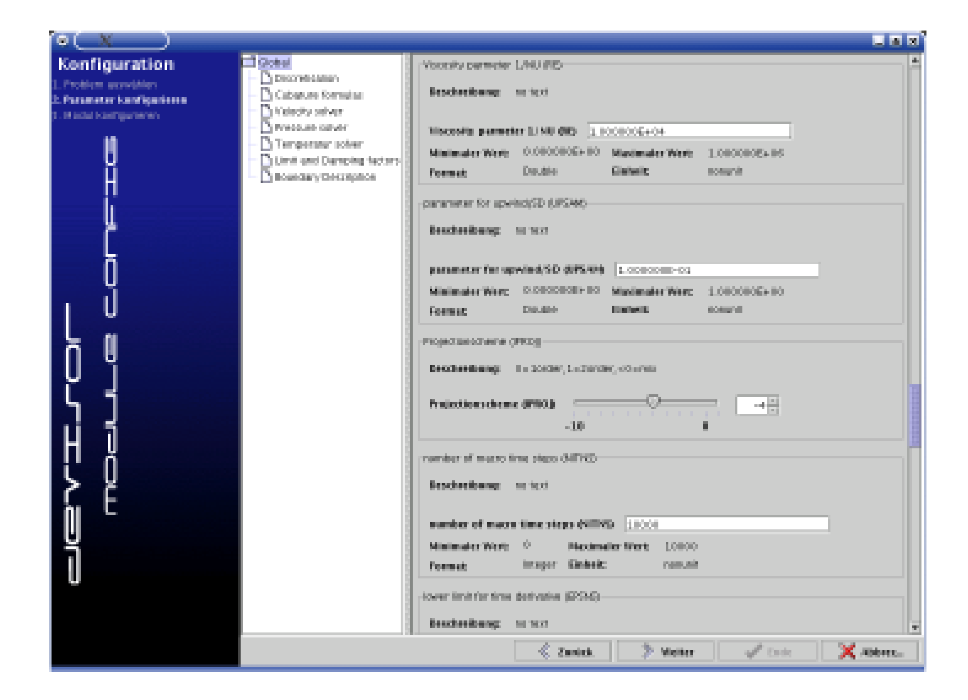


Abbildung A.12: Schritt 2 – Konfiguration

sammengesetzt, deswegen werden alle möglichen Parameterpanels im folgenden einzeln vorgestellt. Was für ein Parameter genau in einem Parameterpanel eingestellt wird, kann immer aus der gegebenen Beschreibung und dem Parameternamen entnommen werden.

Die Parameterpanels

Im folgenden werden im einzelnen die verschiedenen Parameterpanels vorgestellt.

Textfelder

Es kann ein oder mehrere Textfelder geben. Falls nur ein Textfeld gegeben ist, sind auch minimaler Wert, maximaler Wert, Format und Einheit gegeben. Bei Format wird das Format einer zulässigen Eingabe angegeben, möglich sind hier **Integer**, **Double** und **String**, also nur ganze Zahlen, reelle Zahlen oder ganz normaler Text. Der minimale und maximale Wert stellen eine obere und untere Grenze für die zulässige Eingabe dar, falls es sich bei der Eingabe um eine Zahl handeln muß. Wenn mehrere Textfelder gegeben sind, wird nur das Format angegeben. Falls die Eingabe unzulässig ist, gibt es bei Wechseln des Knotens auf der linken Seite der dynamischen GUI eine Fehlermeldung. Das gleiche passiert, wenn man in der dynamischen GUI einen Schritt vor oder zurück gehen will. Ein Beispiel für ein Textfeld ist in Abb. A.13 zu sehen.

Checkboxen

Hier können ein oder mehrere Checkboxen gegeben sein. Mit einem Häkchen wird die Auswahl bestätigt. Es ist eine Mehrfachauswahl möglich.

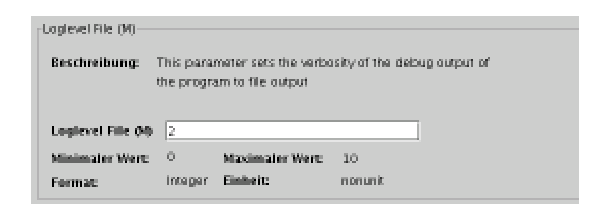


Abbildung A.13: Textfelder

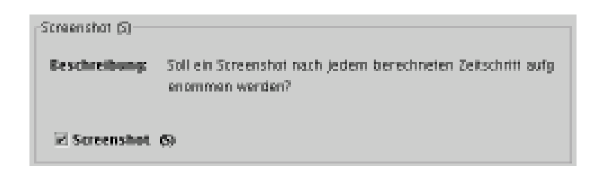


Abbildung A.14: Checkboxen

Radiobuttons

Es kann nur einer der Radiobuttons jeweils ausgewählt werden, womit nur eine Einzelauswahl möglich ist.



Abbildung A.15: Radiobuttons

Slider

Es sind ein oder mehrere Slider möglich. Der Wert kann entweder durch Verschieben des Sliders mit der Maus oder durch Verändern des Spinnerwertes manipuliert werden. Im Spinner wird der genaue Sliderwert angezeigt. Es sind nur **Integer**-Werte möglich.



Abbildung A.16: Slider

Farbauswahl

Durch Betätigen des ”`Wähle Farbe`”-Buttons erscheint ein Standardfarbauswahldialog, in dem eine Farbe gewählt werden kann. Die gewählte Farbe wird in dem Farbfeld angezeigt.



Abbildung A.17: Farbauswahl

Auswahl von Parametern

Für die Auswahl von beliebigen Parametern gibt es vier verschiedene Parameterpanels, zwei davon sind in Abb. A.18 und Abb. A.19 zu sehen. Mit dem in Abb. A.18 dargestellten Parameter ist eine Mehrfachfachauswahl von Parametern möglich. Die linke Liste enthält alle zur Auswahl stehenden Parameter, die Liste auf der rechten Seite enthält hingegen alle bereits ausgewählten Parameter, wobei hier Parameter mehrfach ausgewählt werden können. Hier eine Übersicht über die Buttons:

→	Der in der linken Liste ausgewählte Parameter wird der rechten Liste hinzugefügt.
←	Der in der rechten Liste ausgewählte Parameter wird aus der rechten Liste entfernt.
↑	Der in der rechten Liste ausgewählte Parameter wird um eins nach oben verschoben.
↓	Der in der rechten Liste ausgewählte Parameter wird um eins nach unten verschoben.
”`Einstellungen...`”	Für den in der rechten Liste ausgewählten Parameter wird ein Dialog geöffnet, in dem der Parameter konfiguriert werden kann.

Zusätzlich kann man den Konfigurationsdialog für einen Parameter aus der rechten Liste öffnen, indem man auf diesen Parameter doppelklickt. In der unteren linken Seite wird die Anzahl der ausgewählten Parameter angezeigt. Es ist immer eine obere und untere Schranke für die zulässige Anzahl angegeben. Sobald die Auswahl sich außerhalb der erlaubten Grenzen befindet, wird sie zur Warnung rot angezeigt. Solange die Auswahl außerhalb der erlaubten Grenzen ist, kann weder der Knoten im Baum gewechselt werden noch kann man einen Schritt vor oder zurück gehen. In dem Parameter aus Abb. A.19 ist nur die Auswahl eines einzigen Parameters möglich, der über den Button ”`Einstellungen...`” konfiguriert werden kann. Ansonsten gibt es Checkboxen und Radiobuttons als mögliche Auswahlelemente. Bei Checkboxen können mehrere Parameter einmalig gewählt werden, bei Radiobuttons kann nur ein Parameter gewählt werden. Die ausgewählten Parameter können auch dort über ”`Einstellungen...`”-Buttons konfiguriert werden.

Auswahl einer FEAST-Datei

Mit dem in Abb. A.20 abgebildeten Parameter kann eine FEAST-Datei ausgewählt werden, z.B. für den Aufruf von GRID oder FEATFLOW. Mit ”`Lade Domain`” wird ein Standarddatei-

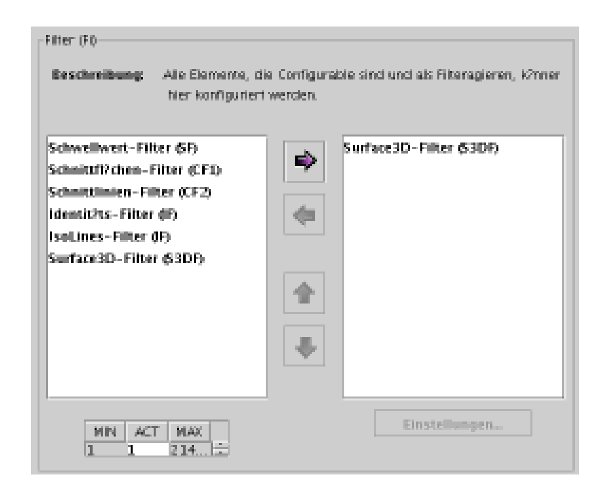


Abbildung A.18: Mehrfachauswahl von Parametern

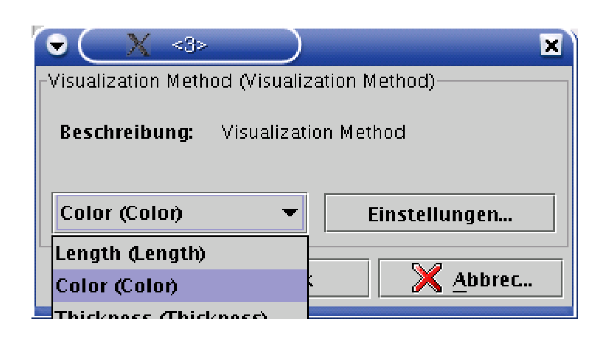


Abbildung A.19: Einfachauswahl

auswahldialog gestartet, in dem eine FEAST-Datei ausgewählt werden kann. Die ausgewählte FEAST-Datei kann man sich auch ansehen, indem man den "`Zeige Domain`"-Button drückt. Dann erscheint der in Abb. A.21 abgebildete Dialog. Die Funktionen des FEAST-Betrachters sind eingeschränkt, aber es ist möglich, über die in der Menüleiste gegebenen Icons zu zoomen und mit der Maus die Domain zu verschieben.



Abbildung A.20: Auswahl einer FEAST-Datei

Eingabe von Funktionen

Mit dem in Abb. A.22 gegebenen Parameter ist es möglich, Funktionen einzugeben. Das wird z.B. in FEATFLOW genutzt. Die obere Tabelle enthält defaultmäßig vorgegebene Funktionsparameter, die um eigene Parameter mittels "`Parameter hinzufügen`" ergänzt werden können. Dann erscheint am Ende der Tabelle eine editierbare neue Reihe. Mit "`Parameter löschen`" hingegen kann ein ausgewählter Funktionsparameter wieder gelöscht werden.

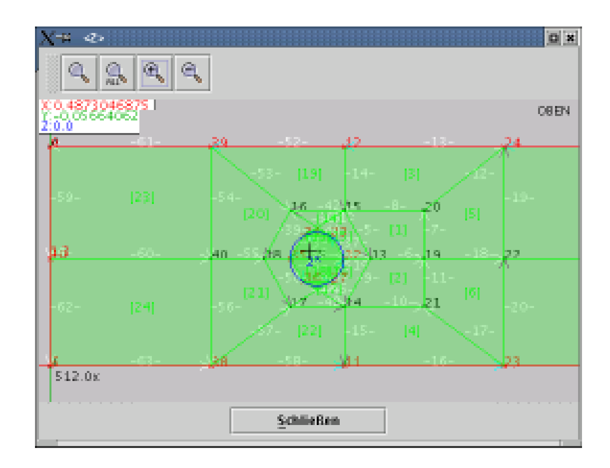


Abbildung A.21: Ein FEAST-Betrachter

Defaultmäßig vorgegebene Funktionsparameter können hingegen weder editiert noch gelöscht werden, was durch die roten und grünen Punkte in der Spalte "`Löschbar`" angedeutet wird. Die Funktion selbst kann in dem unten gegebenen Textfeld eingetragen werden.

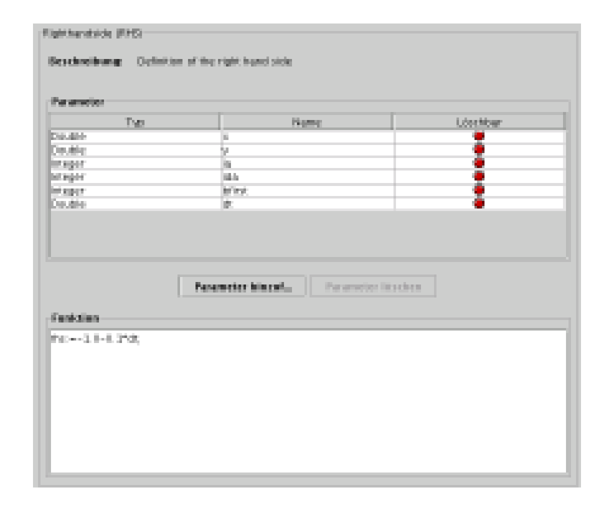


Abbildung A.22: Eingabe von Funktionen

Datenaustausch zwischen Modulen

Mit dem in Abb. A.23 abgebildeten Parameter ist es möglich, ein Numerikmodul auszuwählen und von diesem Daten anzufordern. Die Auswahl eines Moduls erfolgt über den Button "`Wähle Modul`". Dann erscheint ein Modulauswahldialog ähnlich dem in Abb. A.3 mit dem einzigen Unterschied, daß in der Modulliste für einen ausgewählten Server die laufenden und inaktiven Module angezeigt werden. Das ausgewählte Modul wird dann angezeigt und es kann ein Ergebnis ausgewählt werden, falls Ergebnisse bei dem ausgewählten Modul verfügbar sind. Die Ergebnisse sind in der Combobox gegeben. Über "`Einstellungen`" wird der in Abb. A.24 dargestellte Dialog geöffnet. Hier kann man die gewünschten Daten für das ausgewählte Ergebnis einstellen: es ist möglich, ein Datenfeld und Zeitschritte auszuwählen. Für

die Datenfeldauswahl ist eine Combobox gegeben. Da die Anzahl der Zeitschritte sehr groß sein kann (10.000 und mehr), hat man mehrere Möglichkeiten, um Zeitschritte auszuwählen. Mit ”`Gehe`” ist es möglich, zu einem Zeitschritt in der Liste zu springen, was besonders bei einer großen Anzahl von Zeitschritten nützlich sein kann. Für die Angabe eines Zeitschritts hat man generell immer zwei Möglichkeiten: entweder man gibt die Position des Zeitschritts in der Liste an oder die Zeitschrittnummer. Bei ”`Position`” wird die Eingabe des Zeitschritts als die Position des gewünschten Zeitschritts interpretiert. Gezählt wird von 1 an. Bei ”`Wert`” wird die Eingabe als Zeitschrittnummer interpretiert. Falls es diesen Zeitschritt nicht gibt, wird entweder der nächsthöhere oder der nächstniedrigere Wert gewählt, was über einen Togglebutton eingestellt werden kann. Der Togglebutton kann die zwei Zustände ”`F`” und ”`C`” für ” Floor” und ” Ceil” annehmen, es wird also entweder hoch- oder abgerundet. Falls sich der eingegebene Wert außerhalb der angegebenen Grenzen befindet, wird immer der höchste oder niedrigste Wert genommen. Für die Auswahl von Zeitschritten gibt es nun zwei Möglichkeiten, entweder per Maus oder per Textfeldeingabe. Die Eingabe über Textfeld eignet sich besonders für große Intervalle. Das Intervall wird dazu in den Feldern ”`Von`” und ”`Bis`” eingetragen. Die Schrittweite kann über einen Spinner eingestellt werden, d.h. bei z.B. Schrittweite fünf wird nur jeder fünfte Zeitschritt ausgewählt. Für die Angabe des Intervalls kann man wieder zwischen ”`Wert`” und ”`Position`” wählen. Über die Buttons” `Wähle an`”, ”`Wähle ab`” und ”`Invertieren`” kann entschieden werden, ob die markierten Zeitschritte wirklich ausgewählt, abgewählt oder auswählen/abwählen jeweils ausgetauscht werden soll. Das gleiche ist auch mit der Maus möglich. Dazu muß man einfach mit der Maus ein Intervall markieren und die rechte Maustaste betätigen, damit das in Abb. A.24 abgebildete Popup-Menü erscheint. In dem Popup-Menü stehen wieder die Möglichkeiten ”`Wähle an`”, ”`Wähle ab`” und ”`Invertieren`” zur Verfügung. Zusätzlich werden die Position und der Wert des markierten Intervalls angegeben. Um zwischen lediglich markierten und wirklich ausgewählten Zeitschritten unterscheiden zu können, sind markierte Zeitschritte lila, wirklich ausgewählte Zeitschritte rosa und nicht ausgewählte Zeitschritte weiß.

Schritt 3 – Konfiguration des Moduls

Hier wird nun das Modul unabhängig vom ausgewählten Problem konfiguriert. Der Aufbau der Oberfläche ist ja schon in Schritt 2 beschrieben worden. Erst im dritten Schritt wird der ”`Ende`” Button aktiv und damit kann erst hier mit Drücken dieses Buttons die Konfiguration erfolgreich abgeschlossen werden.

A.5 GRID — Ein Gittereditor für zwei- und dreidimensionale Gitter

A.5.1 Einleitung

GRID ist ein 3D und 2D Gittereditor zur Erstellung von Basisgittern für die Verwendung in numerischen Simulationsprogrammen wie z.B. FEATFLOW.
GRID ist Teil des DEVISOR-Paketes, kann aber auch unabhängig davon zur Generierung von 3D und 2D-Gitterdateien verwendet werden. Der Gittereditor unterstützt momentan lediglich das Speichern im FEAST-Format, da das die Hauptanforderung an den Editor war.
Der Editor unterstützt unterschiedliche Typen *Finiter Elemente*:



Abbildung A.23: Wahl eines Numerikmoduls

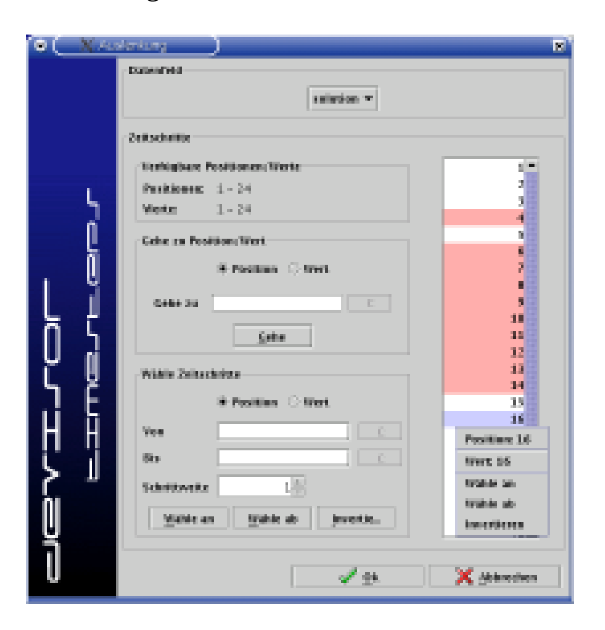


Abbildung A.24: Zeitschrittauswahl

Tris Dreiecke im zweidimensionalen Raum,

Quads Vierecke im zweidimensionalen Raum,

Tetras Tetraeder im dreidimensionalen Raum,

Hexas Hexaeder im dreidimensionalen Raum.

Um den Raum zu beschreiben, den die Finiten Elemente füllen, werden sogenannte *Boundaries* benutzt. Die vom Editor unterstützten Boundary-Typen sind:

SegmentList Ein Polygon im zweidimensionalen Raum. Die Umlaufrichtung gibt dabei an, ob die SegmentList einen Bereich umschließt oder ob sie einen Bereich ausschließt.

Circle Ein Kreis im zweidimensionalen Raum. Auch hier entscheidet die Umlaufrichtung, ob ein Bereich umschlossen wird, oder ob er ausgeschlossen wird.

Sphere Eine Kugel im dreidimensionalen Raum.

Cubicle Ein Quader im dreidimensionalen Raum.

Triangulation Ein durch ein Dreiecksnetz beschriebenes Objekt im dreidimensionalen Raum. Eine Triangulation kann in der aktuellen Version des Editors nicht direkt erstellt werden, sondern nur durch Importieren von WaveFront-Dateien.

Das Erstellen der Objekte geschieht mit Hilfe der Maus oder durch Dialoge. Die Dialoge ermöglichen insbesondere die exakte Eingabe von Zahlenwerten.
Der Editor besitzt zwei unterschiedliche Betriebsmodi:

- Im 2D-Modus gibt es ein einzelnes großes Editorfenster. Alle Werkzeuge zum Erstellen dreidimensionaler Objekte sind gesperrt.

- Im 3D-Modus ist das Fenster in vier unterschiedliche Bereiche aufgeteilt, wie man es aus den bekannten 3D-Editoren wie Maya oder Blender kennt. Die Teilbereiche enthalten die Front-, Side- und Top-Ansicht und zusätzlich eine 3D-Ansicht, in der sich die Ansicht frei bewegen läßt.

Da GRID ein Teil des DEVISOR-Pakets ist, ist es möglich, den Editor vom Kontrollmodul aufzurufen und die Ergebnisse direkt zur weiteren Bearbeitung weiterzuvermitteln.

A.5.2 Installation

Die Installation von GRID wird in Kapitel A.2 dieses Handbuches beschrieben.
Die Einzelversion von GRID lässt sich mit dem Befehl

```
javac -classpath devisor/log4j-1.2.8.jar:.
       devisor/grid/main/GridApp.java
```

aus den Quelltexten übersetzen und mit

```
java -classpath devisor/log4j-1.2.8.jar:.
      devisor.grid.main.GridApp
```

starten.

A.5.3 Die Komponenten des Editors

Der Editor besteht aus folgenden Komponenten:

- dem Menü (s. Kapitel A.5.3),
- den Werkzeugleisten (s. Kapitel A.5.3),
- dem Hauptfenster bestehend aus AUFSICHT, SEITENANSICHT, und VORDERANSICHT und 3D-Ansicht im 3D-Modus oder nur aus der OBEN-Ansicht im 2D-Modus (s. Kapitel A.5.3),
- die Statuszeile (s. Kapitel A.5.3).

Das Hauptfenster

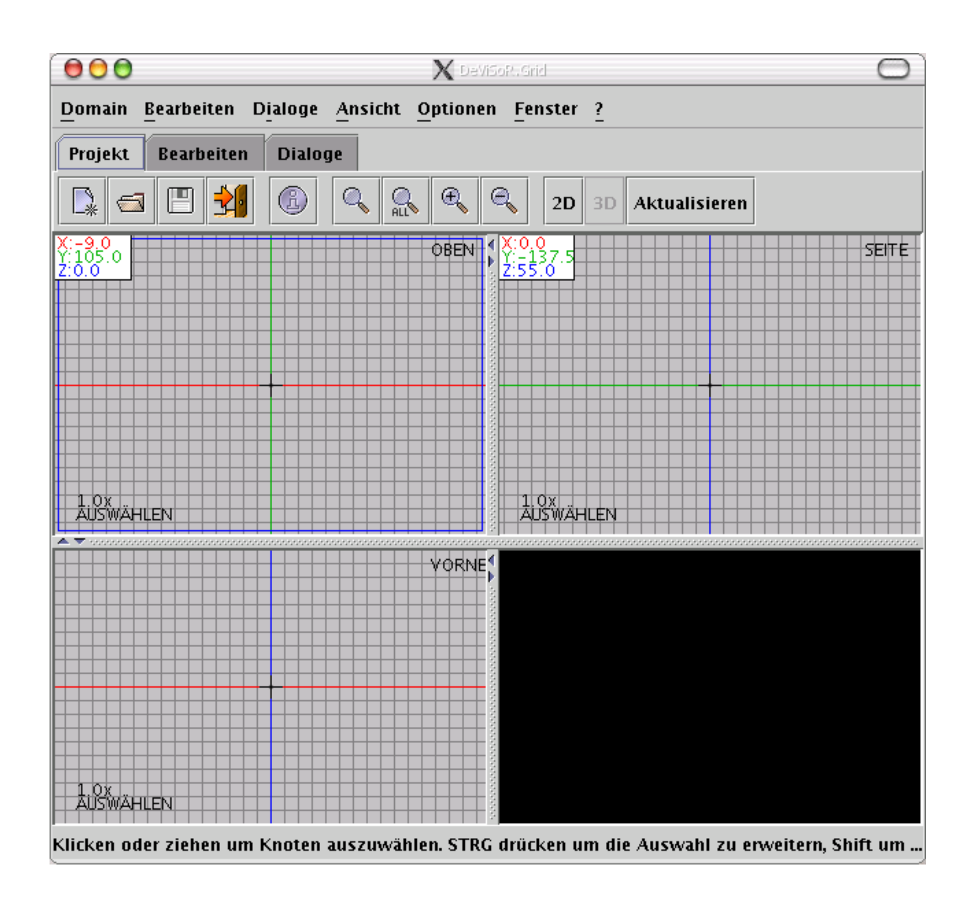


Abbildung A.25: Das Hauptfenster

Das Hauptfenster enthält im 2D-Modus nur die OBEN-Ansicht. Mit der Maus können hier je nach eingestelltem Maus-Modus Finite Elemente und Boundaries erstellt werden.
Im 3D-Modus zeigt das Hauptfenster vier Ansichten. Änderungen in einer Ansicht wirken sich direkt auf die anderen Ansichten aus.

Die Ansichten AUFSICHT, SEITENANSICHT, VORDERANSICHT In der linken oberen Ecke werden die absoluten Koordinaten angezeigt, die der aktuellen Mausposition entsprechen. Die dritte Koordinate, die keine direkte Beziehung zur Mausposition hat, kann verändert werden, indem man die rechte Maustaste gedrückt hält und die Maus nach oben bzw. unten verschiebt. Ein Linksklick in eine Ansicht aktiviert diese. Funktionen wie das Zoomen wirken sich immer auf die aktive Ansicht aus. Alle weiteren Auswirkungen der Maus hängen vom ausgewählten Mausmodus ab. Die 3D-Ansicht liefert im allgemeinen kein direktes Feedback. Da es besonders auf "langsamen" Rechnern bei großen Modellen leicht zu Performanz-Problemen kommen kann, wird die 3D-Ansicht nur aktualisiert, wenn der `[Aktualisieren]`-Knopf in der Toolbar gedrückt wird.
In den Ansichten mit fester Kameraposition (OBEN, SEITE, VORNE) sieht man ein Gitternetz. Die Standardgröße des Gitters ist auf 10 gesetzt. Farbige Linien kennzeichnen die Achsen. Ein kleines Kreuz kennzeichnet den Mittelpunkt des Fensters.

Die 3D-Ansicht Die 3D-Ansicht läßt sich nur über die Maus steuern. Bei gedrückter linker Maustaste rotiert die Kamera um den Nullpunkt; bei gedrückter mittlerer Maustaste kann man in die angezeigte Domain hineinzoomen bzw. herauszoomen; bei gedrückter rechter Maustaste verändert man die Position der Kamera (siehe Kapitel A.5.3).
Die Arbeitsfenster sind abgegrenzt durch Balken, an denen kleine Pfeile angebracht sind. Diese erleichtern die Arbeit, denn man kann mit ihnen die Arbeitsfenster in Pfeilrichtung maximieren. Genauer ausgedrückt: wenn man auf eins der Pfeile am Rand des Arbeitsfensters mit der linken Maustaste klickt, das vom Fenster weg zeigt, wird das Fenster in die Richtung des Pfeils maximiert. Wenn man dann den anderen Pfeil in die entgegengesetzte Richtung drückt, wird die vorherige Ansicht wierderhergestellt. Drückt man wieder auf den Pfeil zum Fenster hin, wird dieses nun von dem anderen überdeckt. Ähnliche Fenstermanipulationen können in der Menüleiste unter dem Menü **Windows** vorgenommen werden.

Das 3D-Fenster Obwohl das schwarze Voransichtsfenster unten rechts in der Ecke zusammen mit den Arbeitsfenstern angeordnet ist, ist es kein Editierfenster, sondern ein dreidimensionaler Betrachter der erstellten Objekte.
Wenn man die rechte Maustaste hält und die Maus verschiebt, verschiebt sich das Objekt in der zum Fenster parallelen Ebene. Wenn man die mittlere Maustaste hält, dann verschiebt sich das Objekt in der zum Fenster senkrechten Ebene, d.h. wenn man nach vorne mit der Maus fährt, dann entfernt sich das Objekt vom Betracher, wenn man zurück fährt, dann kommt das Objekt dem Betrachter entgegen, bis es hinter ihm aus dem Sichtfeld verschwindet.
Hinzu kommt noch die gleiche Vorgehensweise bei der linken Maustaste, um die Objekte zu drehen.

Die Toolbars

Die Toolbars sind in verschiedene Gruppen aufgeteilt. Die erste Gruppe enthält alle Funktionen, die sich auf die aktuelle Domain beziehen, also Laden, Speichern, Importieren, Beenden, usw. Die zweite Gruppe enthält Funktionen, mit deren Hilfe man die Domain manipulieren kann. Neben Funktionen wie Kopieren, Einfügen und Löschen hat man hier auch die Möglichkeit festzulegen, in welchem Modus sich die Maus befindet (z.B. Selektieren, Punkte einfügen, usw.). Die dritte Gruppe ermöglicht es, Dialoge für alle Objekte aufzurufen, die eine Domain

enthalten kann. Die Dialoge ermöglichen es dem Benutzer u.a. die Koordinaten der Punkte exakt als Gleitkommazahlen einzugeben.
Im folgenden werden die Namen der Werkzeuge aufgelistet, die in den Toolbars vorkommen. Sie entsprechen den Icons auf den ensprechenden Bildern von links nach rechts. Da deren Funktionalität genau den zugehörigen Menüpunkte in der Menüzeile entspricht, werden sie hier nicht weiter erläutert, sondern weiter unten bei der Menüleiste ausführlich erläutert.

Die Werkzeuge der Projekt-Toolbar



Abbildung A.26: Die Projekt-Toolbar

Neue Domain, Domain laden, Domain speichern, Beenden, Domain durchsuchen, Zoom zurücksetzen, Alles zeigen, Hineinzoomen, Herauszoomen, 2D-Modus, 3D-Modus, Aktualisieren

Die Werkzeuge der Edit-Toolbar



Abbildung A.27: Die Edit-Toolbar

Auswahl kopieren, Kopierte Auswahl einfügen, Ausgewählte Objekte löschen, Ansicht verschieben, Objekte auswählen, Ausgewählte Objekte verschieben, Ausgewählte Objekte rotieren, Ausgewählte Objekte skalieren, Punkte (Nodes) erzeugen, Kanten (Edges) erzeugen, Tris erzeugen, Quads erzeugen, SegmentLists erzeugen, Kreise (Circles) erzeugen, Quader (Cubicles) erzeugen, Kugel (Spheres) erzeugen, Zylinder (Cylinders) erzeugen

Die Werkzeuge der Dialogs-Toolbar



Abbildung A.28: Die Dialogs-Toolbar

Knotendialog anzeigen (Nodes), BoundaryNode-Dialog anzeigen, BoundaryNode3D-Dialog anzeigen, Kantendialog anzeigen (Edges), Tridialog anzeigen, Quaddialog anzeigen, Tetradialog anzeigen, Hexadialog anzeigen, Kreisdialog anzeigen (Circles), Quaderdialog anzeigen (Cubicles), Kugeldialog anzeigen (Spheres), Zylinderdialog anzeigen (Cylinder)

Die Statuszeile

Die Beschriftung der unteren Zeile des Hauptfensters zeigt Hinweise zur Benutzung der ausgewählten Werkzeuge und weist auf veränderte Statusinformationen hin. Sie gibt an, ob eine bestimmte Aktion gelungen ist oder nicht.

Die Menüleiste

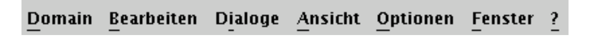


Abbildung A.29: Die Menüleiste

Unter der Hauptfenstertitelleiste befindet sich die Menüleiste. Diese besteht aus sieben Menüpunkten: `[Domain]`, `[Bearbeiten]`, `[Dialoge]`, `[Ansicht]`, `[Optionen]`, `[Fenster]` und `[?]`.

Der Menüpunkt `[Domain]`



Abbildung A.30: Das Domain-Menü

In diesem Menüpunkt können allgemeine Aktionen bezüglich der Domain ausgeführt werden. Dabei ist mit der Domain alles gemeint, was an Gitterelementen erstellt worden ist.

Mit dem Menüeintrag **Neue Domain** wird eine neue Domain erzeugt. Die alte Domain wird gelöscht, falls nicht vorher gespeichert wurde.

Der Menüeintrag **Öffnen** öffnet einen Standard-Dateiauswahldialog. Hier kann man bestimmen, welche Datei nun als Domain geladen werden soll. Geöffnet werden können Dateien im **FEAST**-Dateiformat (die Dateinamen müssen auf .feast enden).
Dieser Menüeintrag entspricht dem Button `[Öffnen]` in der Werkzeugleiste **Main**.

Der Menüeintrag **Speichern** öffnet einen Standard-Dateiauswahldialog. Hier kann man bestimmen, unter welchem Dateinamen und im welchen Format die Domain gespeichert werden soll. Wie der Standard-Speicherdialog aussieht und zu bedienen ist, kann an den entsprechenden Stellen nachgelesen werden.
GRID speichert die Domain im **FEAST**-Dateiformat. Aus dem 2D-Modus werden nur die 2D-Objekte der Domain gespeichert (und etwa Cubicles ignoriert, die nur in 3D-Modellen existieren), aus dem 3D-Modus nur 3D-Objekte (hier werden etwa SegmentLists nicht gespeichert).

Mit dem Menüeintrag **Importieren** lassen sich Objekte in die aktuelle Domain laden. Dabei hat man die Möglichkei außer den Standard-FEAST-Dateien auch Java `.obj`-Dateien als *BoundaryTriangulation* zu laden. Wiederum erscheint ein Standard-Dateiauswahldialog, der hier nicht weiter dokumentiert wird. Das neu geladene Objekt ist markiert, so daß man bequem die Lage, Größe und Position der Auswahl verändern oder leicht kopieren kann (siehe Kapitel A.5.3).

Mit dem Menüeintrag **Drucken** kann die aktuelle Domain ausgedruckt werden.Es erscheint ein systemeigener Druckdialog, mit dem man die Druckeinstellungen vornehmen kann. Dieser Dialog unterscheidet sich von System zu System und auch von Drucker zu Drucker. Deswegen kann die genaue Bedienung des Dialogs in der entsprechenden System-Quelle nachgelesen werden.

Der Menüeintrag **Domain durchsuchen** ist eine gute Hilfsfunktion, um die Objekte und deren Eigenschaften in einer überichtlichen Form zu betrachten. Es öffnet sich ein Fenster mit der aus diversen Explorerfenstern bekannten Baumstruktur. Hier gewinnt man Einblick in den hierarchischen Aufbau der Domain, die aus ElementNodes, BoundaryNodes, BoundaryNodes3D, Edges, Tris, Quads, Tetras, Hexas und Boundaries besteht. Die Elemente in der Liste, die weitere Elemente enthalten, sind mit einem Ordnersymbol gekennzeichnet, die anderen, also Elemente, die keine Unterelemente haben, werden als ein Blatt Papier gezeichnet, dessen rechte obere Ecke umgeknickt ist.
Links von den Ordnersymbolen befindet sich jeweils eine umgedrehte Lupe. Mit dem Klick der linken Maustaste darauf läßt sich Einsicht in die inneren Elemente dieses Ordners gewinnen, die wiederum Ordner sein können. So kann man Schicht für Schicht die betreffenden Ordner ausklappen und die darin enthaltenen Informationen betrachten. Auf der untersten Ebene dieser Hierarchie befinden sich die genauen Koordinaten und die Eltern (Parents) der Elemente. Mit den Eltern sind alle Elemente einer Hierarchieebene höher gemeint, die das aktuelle Element teilen, z.B. wenn ein Knoten gleichzeitig der Startpunkt einer Kante ist und der Endpunkt einer anderen, dann sind die beiden Kanten Eltern des Knotens. Diese Eltern-Information ist vor allem wichtig für das Löschen der Elemente (siehe unten). Denn, wenn ein Knoten, zu dem mehrere Kanten adjazent sind, die zu mehreren Quad-Elementen gehören, und die wiederum zu mehreren Hexas gehören, gelöscht wird, werden auch all die abhängigen Elemente gelöscht. Diese unumgängliche Eigenschaft führt dazu, daß manchmal eine einzige Operation weitreichende Folgen haben kann.

Weiterhin ist diese Ansicht wichtig, um z.B. in die in den Arbeitsfenstern nicht immer sichtbare Elementennummern Einsicht zu haben, damit man in den ensprechenden Dialogen die richtige Nummer eingibt (siehe unten).
Links oben in dem Domain-Overview-Fenster befindet sich eine Update-Checkbox, die bei dem entsprechenden Mausklick darauf die aktuellen Veränderungen auf der Zeichenfläche in die Baumstruktur übernimmt. So kann man dieses Fenster ständig parallel zum Hauptfenster offenhalten und hat trotzdem die aktuellste Übersicht.
Wenn man nun nicht mehr diese Übersicht haben möchte, so drückt man den Button `[Ok]`, um diesen Dialog zu verlassen.
Dieser Menüpunkt entspricht dem Button `[Domain Overview]` in der Werkzeugleiste **Main**.

Das Problem bei den Domains ist, daß man sie in einem beliebigen Maßstab erstellen kann. Damit man Domains vergleichen, besser verarbeiten oder einfach im gleichen Maßstab hat, existiert der Menüeintrag **Domain normalisieren**. Dieser verkleinert, bzw. vergrößert und verschiebt die Domain so, daß sie genau in den Bereich [-1,1] auf allen drei Koordinatenachsen paßt.
Zum Schluß kommt der Menüeintrag **Exit**, mit dem das Programm verlassen wird, nicht ohne eine entsprechende Abfrage, ob man die nicht gespeicherten Informationen verlieren möchte.

Der Menüpunkt `[Bearbeiten]`

Unter diesem Menüpunkt befinden sich diverse Werkzeuge, um mit Hilfe der Maus Objekte zu selektieren, bewegen, drehen, kopieren, löschen, eingeben und zu verändern. All diese Werkzeuge entsprechen den Werkzeugen in der Bearbeiten-Toolbar.
Das Werkzeug **Kopieren** macht eine interne Kopie der ausgewählten Elemente (in dieser Version können es nur Knoten und Kanten sein). Die Auswahl geschieht mit dem Werkzeug **Auswählen**, welches weiter unten erklärt wird. Wenn nichts ausgewählt ist, so ist auch die Kopie leer. Zunächst sieht man gar nichts von den Auswirkungen dieses Werkzeugs.

Wenn man aber nun das Werkzeug **Einfügen** mit der linken Maustaste anklickt, so wird die vorher kopierte Auswahl kopiert und ein Stück weiter links, hinten und oben eingefügt. Was mit einem “Stück” gemeint ist, hängt von der aktuellen Zoom-Stufe(siehe unten) ab. Es erscheint auch ein kleiner Dialog, in dem man genau eingeben kann, wo die Kopie plaziert werden soll.
Es kann manchmal vorkommen, daß das Einfügen mißlingt, weil an einer der neuen Stellen schon ein Knoten vorhanden ist. Dies läßt sich leicht beheben, indem man das Einfügen nochmal versucht und dabei in dem Dialog einen anderen Wert für die Positionierung der Kopie eingibt.

Für das Löschen existiert das Werkzeug **Löschen**. Damit lassen sich ausgewählte Knoten und die abhängigen Objekte löschen.

Danach folgt der Menüeintrag **Ansicht verschieben**. Wenn dieser mit der linken Maustaste ausgewählt wurde, dann kann man das Sichtfenster verschieben, und dadurch Einsicht in die bisher unsichtbaren Teile des Gesamtgitters gewinnen. Um das zu erreichen, plaziert man den Mauszeiger an eine beliebige Stelle im aktuellen Fenster, drückt die linke Maustaste und hält sie gedrückt, während man die Maus in die zu dem freizulegenden Teil des Raums entgegen-

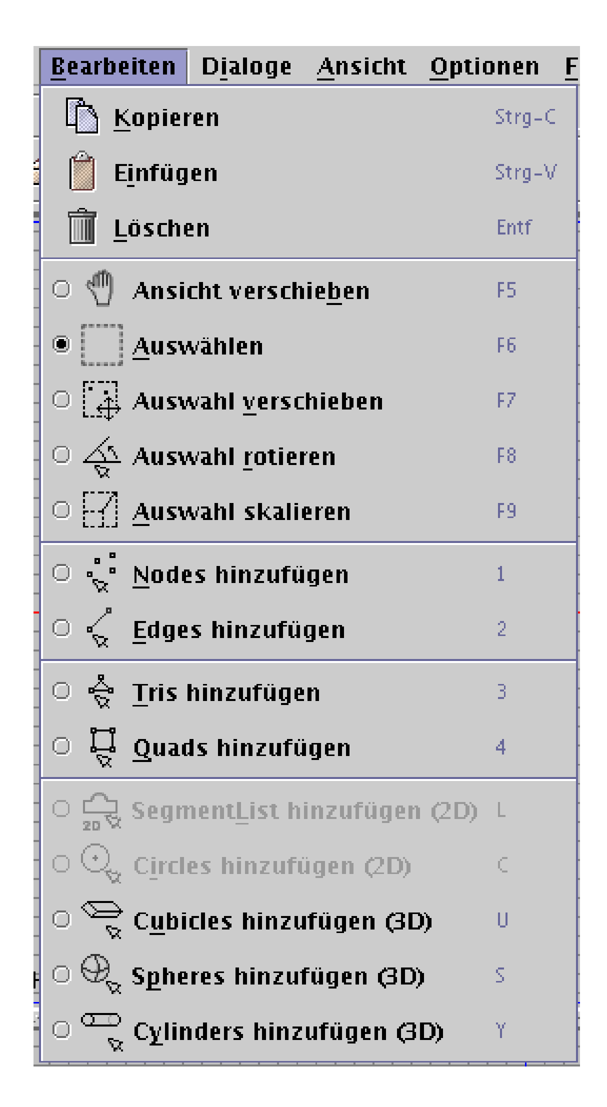


Abbildung A.31: Das Bearbeiten-Menü

gesetzte Richtung schiebt, also z.B., wenn man wissen will, was sich oberhalb des Fensters befindet, schiebt man die Maus nach unten.

Gleich darauf folgt der Menüeintrag **Auswählen**. Damit kann man diverse Objekte auswählen, um auf der gesamten Auswahl eine Aufgabe durchzuführen. Dabei gibt es implizit zwei Modi, die Boxauswahl und die Einzelauswahl, die auch kombiniert werden können und sollen. Die Boxauswahl ist am einfachsten und schnellsten durchzuführen: man hält die linke Maustaste gedrückt und zieht ein Rechteck um die Fläche, in der sich die zu markierende Objekte befinden. Wenn man die Maustaste losläßt, werden die Knoten der ausgewählten Objekte anders gefärbt als vorher. Das bedeutet, daß diese selektiert sind. Dabei können unerwünschte Objekte in die Auswahl aufgenommen werden, bzw. erwünschte nicht erfastt werden. Dafür gibt es die Einzelauswahl. Um zusätzliche Objekte auszuwählen, drückt man die `CTRL`[1] -Taste und klickt

[1]Auf manchen Tastaturen ist das die `Strg`-Taste oder auch die `Control`-Taste. Im folgenden wird immer die `CTRL`-Bezeichnung verwendet.

direkt auf die entsprechenden Objekte, die zusätzlich in die Auswahl aufgenommen werden sollen. Damit werden auch deren Knoten ausgewählt. Wenn man Objekte wieder entfernen möchte, drückt man die `Shift`-Taste und klickt mit der linken Maustaste auf Objekte, die nicht mehr ausgewählt werden sollen. Diese sind damit nicht mehr ausgewählt.
Man kann direkt die Objekte bearbeiten, wenn mann auf diese doppelklickt. Zunächst erscheint ein kleines Popup-Fenster neben dem Mauszeiger. Dieses enthält alle Elemente, die sich unter dem Mauszeiger befinden. Es können mehrere Objekte in 3D hintereinanderliegen, so daß es nicht klar ist, welches der Benutzer meint. Außerdem, auch wenn man nur auf einen einzigen Knoten klickt, möchte man evtl. die von ihm abhängige Kante oder das von der Kante abhängige Tetra-Objekt bearbeiten. Man sucht sich das Objekt, welches man wirklich meint, aus, und es öffnet sich ein dem Objekt entsprechendes Editorfenster (siehe unten), in dem man genaue Einstellungen bezüglich des ausgewählten Objekts vornehmen kann.
Wenn man die gesamte Auswahl aufheben möchte, so versucht man am Besten einen freien Bereich des Arbeitsfensters auszuwählen.
Wenn man Objekte selektiert hat, hat man eine ganze Reihe von möglichen Operationen, die man auf diese Auswahl anwenden kann: man kann sie verschieben, drehen, skalieren, kopieren und löschen.

Zum Verschieben existiert der Menüeintrag **Auswahl verschieben** direkt unter **Auswählen**. Bevor man diese Funktion nutzen kann, müssen Elemente ausgewählt sein. Nun kann man irgendwo auf die Arbeitsfläche mit der linken Maustaste klicken und ohne loszulassen die Maus verschieben. Zur besseren Übersicht erscheinen zwei dunkle Vierecke, die die alte und die neue Position bestimmen. Wenn man nun die linke Maustaste losläßt, wird die Auswahl in die von der Maus bestimmten Richtung um die von der Maus beim Verschieben zurückgelegten Entfernung verschoben. Dabei sollte der oben ewähnte Trick mit der rechten Maustaste nicht vergessen werden, um in die dritte Koordinate zu schieben.

Das Drehen funktioniert mit dem darunterliegenden Menüeintrag **Auswahl rotieren**. Hier handelt es sich um ein ähnliches Vorgehen wie beim Verschieben. Man hält die linke Maustaste gedrückt und schiebt sie herum. Dabei erscheint um den Ursprung herum ein Kreissegment, der den ausgewählten Winkel grafisch darstellt. Gleichzeitig kann man den eingestellten Winkel als Zahl nahe beim Ursprung ablesen. Es wird, wie man leicht sieht, immer um den Ursprung gedreht. Wenn man nun die Maustaste losläßt, drehen sich alle Elemente in der Auswahl um den eingestellten Winkel.

Das Skalieren kann über den entsprechenden Menüeintrag **Auswahl skalieren** aufgerufen werden. Man kann verschiedene Skalierungen für die verschiedenen Koordinatenrichtungen eingeben, indem man die linke Maustaste wieder gedrückt hält und nach oben und unten verschiebt, um die Skalierung in die Waagerechte und nach links und rechts verschiebt, um die Skalierung in die Senkrechte anzugeben. Es erscheint ein weißes Kreuz, der die Skalierung grafisch verdeutlichen soll, wobei daneben der Skalierungsfaktor mit ausgeschrieben wird.
Um eine in beide Richtungen gleichmäßige Skalierung zu erhalten, hält man beim Verschieben die `Shift`-Taste gedrückt. Es wird ein weißes Kreuz mit gleich großen Armen erzeugt und man skaliert in beide Richtungen gleichmäßig.
Ein Skalierungsfaktor im Bereich $[0, 1[$ verkleinert die Auswahl um diesen Wert, ein Faktor größer als 1 vergrößert die Auswahl. Bei einem Faktor von 1 bleibt die Auswahl unverändert.

Es folgen Werkzeuge, die die direkte Eingabe von grafischen Elementen wie Knoten, Kanten, Tris, Quads, etc. erlauben. Es sind nur die Werkzeuge auswälbar, die in dem ausgewählten Modus (2D bzw. 3D) gültig sind.

Das Werkzeug **Nodes hinzufügen** erlaubt die Eingabe von Knoten. Knoten werden durch ihre Koordinaten beschrieben. Man bewegt die Maus in dem ausgewählten Fenster so lange, bis die Koordinatenanzeige den gewünschten Punkt anzeigt und drückt dann die linke Maustaste. Die Knoten werden durch kleine Punkte dargestellt, an denen deren Nummer steht. Jeder Knoten kann dadurch anhand seiner Nummer referenziert werden.

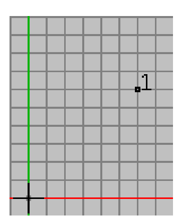


Abbildung A.32: Ein Knoten auf der Zeichenfläche, unausgefüllt

Wenn man versucht, an eine Stelle einen Knoten zu setzen, an der ein schon vorhandener Knoten existiert, wird dies unterbunden, und in der Statusleiste erscheint die Meldung, daß ein Knoten mit ausgewählten Koordinaten schon existiert.
Dieses Werkzeug ist das gleiche wie in der Werkzeugleiste **Bearbeiten**,
`[Nodes hinzufügen]`.

Zu den Knoten gehören auch Kanten. Diese kann man mit dem Werkzeug **Edges hinzufügen** erzeugen. Dies geschieht zunächst analog zu der Knotenerzeugung, denn man definiert einen Start- und einen Endknoten. Es wird also eine gerichtete Kante erzeugt. Die Richtung dieser Kante wird durch einen Pfeil bei dem Endknoten angezeigt. Zusätzlich wird in der Mitte der Kante die Kantennummer eingeblendet. Kantennummern unterscheiden sich von den Knotennummern durch ein vor- und nachgestelltes Minuszeichen, z.B. für die Kante 1 würde dann stehen: -1-.

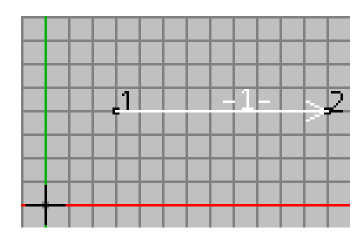


Abbildung A.33: Eine Kante auf der Zeichenfläche

Wenn man in eine einstellbare (siehe unten) Nähe eines existierenden Knotens klickt, dann wird dieser Knoten als ein Knoten der Kante erkannt und übernommen. Wenn dies nicht erwünscht ist, es also ein neuer Knoten erzeugt werden soll, dann muß man bei der Eingabe zusätzlich die `SHIFT`-Taste halten. Ein neuer Knoten wird allerdings nur erzeugt, wenn die neuen Koordinaten mit denen von keinem der existierenden Knoten übereinstimmen. Wenn man es sich mitten

in der Bearbeitung anders überlegt hat und nun doch nicht die Kante zeichnen möchte, dann kann man durch das Klicken der rechten Maustaste dem Programm diesen Wunsch mitteilen. Die Eingabe des Objektes wird sofort unterbrochen. Die eben vorgestellten Prinzipien gelten für alle nachfolgenden Elemente, so daß diese Eigenschaft nicht mehr explizit erwähnt wird. Allgemein gilt, wenn Unterelemente (Knoten, Kanten, Tris und Quads) existieren, können sie ohne weiteres für ein aus diesen Elementen bestehendes Objekt verändert werden.
Dieses Werkzeug ist das gleiche wie in der Werkzeugleiste **Bearbeiten**,
`[Edges hinzufügen]`.

Ein weiteres Werkzeug ist **Tris hinzufügen**. Damit kann man durch die Markierung von drei Punkten mit Hilfe der Maus ein Tri-Objekt erstellen. Da es ein Flächenobjekt ist, wird es ausgefüllt[2].

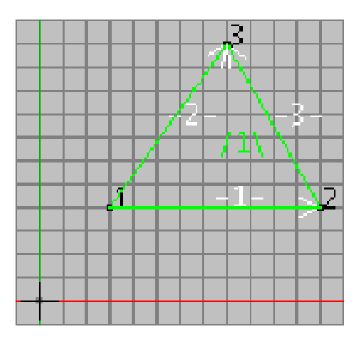


Abbildung A.34: Ein Tri auf der Zeichenfläche

Die Numerierung von Tri-Objekten sieht dann für ein Tri mit der Nummer 1 so aus: /1\.
Dieses Werkzeug ist das gleiche wie in der Werkzeugleiste **Bearbeiten**,
`[Tris hinzufügen]`.

Um die zweite, in GRID mögliche Flächenform, Quads, zu erzeugen, existiert das Werkzeug **Quads hinzufügen**. Genauso wie bei den Tris bestimmt man hier durch Angabe der vier Knoten das Objekt. Die Nummerierung von Quad-Objekten sieht dann für ein Quad mit der Nummer 1 so aus: [1].

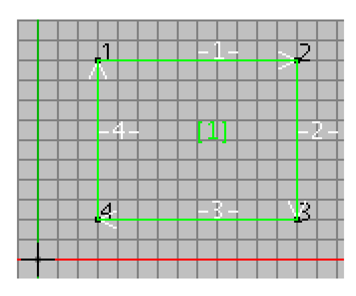


Abbildung A.35: Ein Quad auf der Zeichenfläche

Dieses Werkzeug ist das gleiche wie in der Werkzeugleiste **Bearbeiten**,

[2]Soweit es nicht anders in den Optionen (siehe unten) festgelegt ist.

`[Quads hinzufügen]`.

Die bisher erwähnten Elemente können sowohl in 3D-, als auch in 2D-Domains vorkommen. Die folgenden Boundary-Elemente können nur erstellt werden, wenn sie in dem ihnen zugehörigen Modus aktiv sind. Beim Umschalten der Modi (siehe unten) werden die nicht zu dem Modus gehörende Elemente ausgeblendet (aber nicht entfernt). Dies gilt vor allem beim Unschalten von 2D- in den 3D-Modus.

Das Werkzeug **SegmentList hinzufügen** ist ein 2D-Wekzeug. Damit kann man eine Boundary(Randbeschreibung) vom Typ Boundary-Segment-Liste erstellen. Diese besteht aus einer Menge von Kanten und einer Menge von Kreissegmenten. Dieses Polygon darf nicht offen sein. Man kann wie gewohnt durch Klicken auf die entsprechenden Knotenpositionen die Kanten nacheinander erzeugen, wobei angenommen wird, daß der Endknoten der letzten Kante (des letzten Kreissegments) der Startknoten der nächsten ist. Der Endknoten des letzten Segments muß der Startknoten des ersten sein. Um Kreissegmente im Uhrzeigersinn zu erzeugen, hält man die linke Maustaste gedrückt und zieht in die beliebige Richtung, ein Kreissegment wird mitgezeichnet. Wenn man die Maustaste loslässt, wird das Segment gezeichnet. Wenn man Kreissegmente gegen den Uhrzeigersinn haben möchte, dann hält man bei dem gerade beschriebenen Vorgang zusätzlich die `SHIFT`-Taste fest.

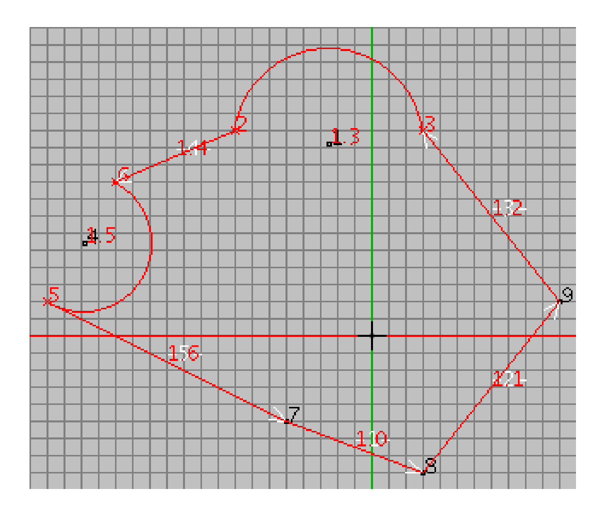


Abbildung A.36: Eine Segmentliste auf der Zeichenfläche

Die Segmentliste wird nicht explizit numeriert, wohl aber ihre Segmente. Die Segmente erhalten zusätzlich zu den Kanten und Knotennummer eine spezielle Segmentnummer, die die Nummer der Liste und die Nummer innerhalb der Segmentliste enthält, getrennt durch einen Dezimalpunkt, z.B. ist das Segment mit der Nummer: “1.1” das erste Segment innerhalb der Segmentliste 1. So können später unmissverständlich die sogenannten Boundary-Nodes zugewiesen werden.
Dieses Werkzeug ist das gleiche wie in der Werkzeugleiste **Bearbeiten**, `[SegmentList hinzufügen]`.

In 2D existiert ein weiteres Randbeschreibungselement, der volle Kreis. Um diese zu erstellen, gibt es das Werkzeug **Circles hinzufügen**. Der Kreis wird anhand von zwei Punkten berechnet,

einmal der Mittelpunkt und einmal der Radius, eingegeben durch einen zweiten Punkt, von dem der Abstand zum ersten Punkt bestimmt wird.

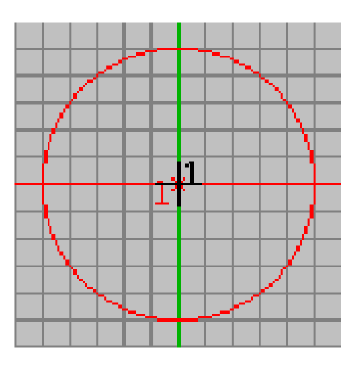


Abbildung A.37: Ein Kreis auf der Zeichenfläche

Wie bei allen Randbeschreibungen kann man auch beim Kreis festlegen, ob er nun bei der Betrachtung vom Rest ausgeschlossen wird (externe Betrachtung), oder ob er in die Betrachtung eingeschlossen wird, und der ganze Rest ausgeschlossen wird. Um das bei der Zeichnung eines Kreises zu berücksichtigen, kann man bei der Eingabe die `CTRL`-Taste drücken, um den äußeren Raum zu beschreiben. Ansonsten wird das Innere des Kreises beschrieben.
Die Nummer des Kreises erscheint neben der Nummer des Mittelpunkts, sie hat allerdings eine andere Farbe als dieser.
Dieses Werkzeug ist das gleiche wie in der Werkzeugleiste **Bearbeiten**, `[Circles hinzufügen]`.

In 3D gibt es drei Arten von Randbeschreibungen: Cubicles(Quader), Spheres(Kugeln) und Cylinders (Zylinder).

Das Werkzeug **Cubicles hinzufügen** fügt einen rechteckigen Quader ein. Mit zwei Punkten wird ein Grundquader erstellt. Die Punkte beschreiben den linken oberen und den rechten unteren Punkt des Quaders. Sie dürfen verständlicherweise nicht auf einer Ebene liegen, deswegen muß bei der Erstellung dieses Objekts die rechte Maustaste verwendet werden, um die dritte Koordinate zu bestimmen. Es muß ein Punkt eingegeben werden, der die Rotation des Quaders bestimmt. Die Rotation kann hier leider sehr unpräzise anhand der simulierten Drehung eingegeben werden. Um eine genaue Eingabe zu erhalten, muß man die 8 Punkte in dem entsprechenden Dialog eingeben. Es existieren für alle Elemente entsprechende Dialoge(siehe unten), mit denen man die Eingabe genauer als mit der Maus vornehmen kann.
Die Nummer des Quaders wird nicht explizit eingezeichnet, wohl aber die Nummern der 8 Eckpunkte, aus denen er besteht.
Dieses Werkzeug ist das gleiche wie in der Werkzeugleiste **Bearbeiten**,
`[Cubicles hinzufügen]`.

Das nächste Werkzeug, **Spheres hinzufügen**, fügt nach dem gleichen Prinzip wie das Werkzeug **Kreise hinzufügen** eine Kugel in die Domain ein.
Auch hier wird nicht explizit, außer der Nummer des Mittelpunkts, die Kugel numeriert.
Dieses Werkzeug ist das gleiche wie in der Werkzeugleiste **Bearbeiten**,

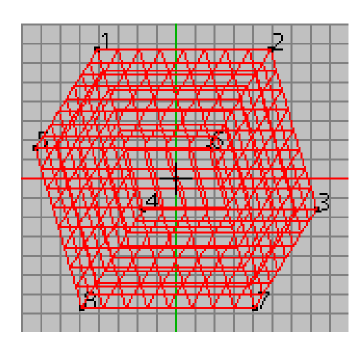


Abbildung A.38: Ein Cubicle auf der Zeichenfläche

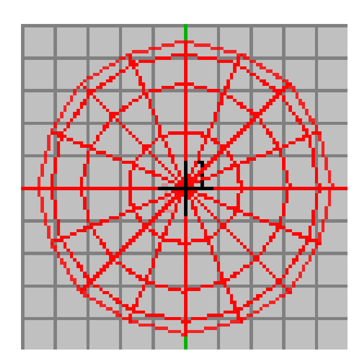


Abbildung A.39: Eine Kugel auf der Zeichenfläche(Top-Ansicht)

`[Spheres hinzufügen]`.

Das letzte Werkzeug unter diesem Menüpunkt ist **Cylinders hinzufügen**. Es fügt einen Zylinder durch die Eingabe von zwei Punkte der Mittelachse und eines dritten Punktes zur Bestimmung des Radius' ein.

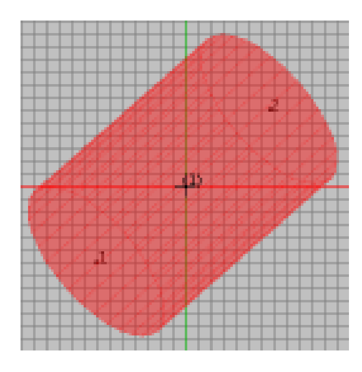


Abbildung A.40: Ein Zylinder auf der Zeichenfläche

In diesem Fall wird die Nummer des Zylinders in der Mitte eingetragen, eingeschlossen durch eine runde Klammer, z.B. wird ein Zylinder mit der Nummer 1 folgendermaßen numeriert: (1).

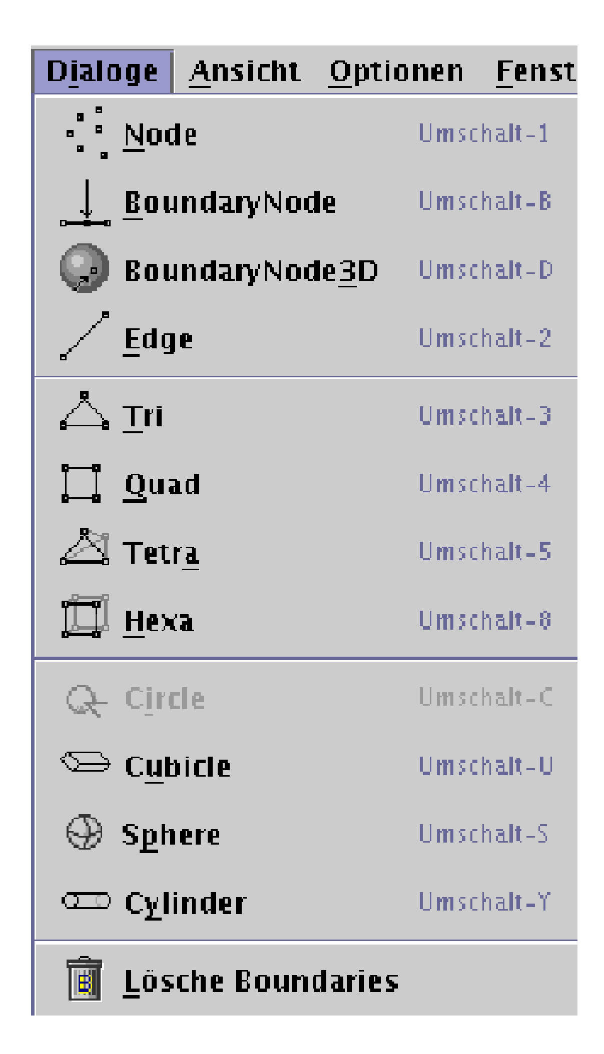


Abbildung A.41: Das Diaolg-Menü

Der Menüpunkt `[Dialoge]`

Über diesen Menüpunkt sowie über die dazugehörigen Buttons in der Werkzeugleiste lassen sich Dialoge öffnen, mit denen die vom Editor unterstützten Objekte neu anlegen, bearbeiten und löschen lassen.[3]
In den Auswahlboxen `[Nummer]` der Dialoge lässt sich festlegen, ob ein neues Objekt des jeweiligen Objekttyps erzeugt oder das Objekt mit der entsprechenden Nummer bearbeitet werden soll. Wurde ein zu bearbeitendes Objekt ausgewählt, so lassen sich nicht in allen Feldern neue Werte setzen, die entsprechenden Felder sind nicht editierbar.[4]
Bestehende Objekte können mit dem Knopf `[Löschen]` aus der Domain entfernt werden, da-

[3]Für die SegmentList existiert kein Dialogfenster. Sie läßt sich nur mit der Maus anlegen. Gelöscht werden kann sie allerdings über den Lösche Boundaries-Dialog.

[4]Wird zum Beispiel ein bestehendes Quad bearbeitet, so lassen sich die Edges, aus denen das Quad besteht, nicht mehr verändern, um sicherzustellen, daß das Quad stets wohldefiniert ist. Die Größe des Quads ließe sich ändern, indem die beteiligten Nodes verschoben werden. Makroeigenschaften lassen sich nur im 2D-Modus ändern, da die Quads im 3D-Modus keine Makroeigenschaften haben.

bei werden auch alle Objekte entfernt, die dieses Objekt enthalten. Mit `[Anwenden]` werden die Werte übernommen, das Dialogfenster bleibt für weitere Eingaben offen. Bei `[OK]` werden die Eingaben übernommen und das Fenster geschlossen, bei `[Abbrechen]` schließt sich das Fenster, ohne die Änderungen durchzuführen. Die einzelnen Eingabemöglichkeiten werden in Kapitel A.5.4 beschrieben.
Zusätzlich zu den schon unter dem Menüpunkt `[Bearbeiten]` genannten Objekten existieren hier auch Dialoge für die Erstellung von Boundary-Nodes und Boundary-Nodes3D. Diese besonderen Punkte werden nicht absolut plaziert, sondern relativ zu einem, ihnen zugehörigen Boundary-Objekt. Dabei wird nach 2D und 3D unterschieden, denn die Punkte werden jeweils anders berechnet. Die Eingabe geschieht in 3D über die entsprechende Boundary und zwei Parameter (Winkel), so daß die endgültige Position anhand einer um den Mittelpunkt der Boundary gelegten Kugel, in die die Boundary vollständig hineinpaßt und dem Schnittpunkt der Linie aus dem Mittelpunkt der Kugel und dem Punkt auf der Kugel, der anhand der zwei Winkel berechnet wird, mit der Boundary berechnet.
In 2D bestimmt man bei einer Segmentliste das Segment und gibt durch einen Wert im Bereich [0,1], wo der Punkt prozentual auf diesem Segment erscheinen soll.

Der Menüpunkt `[Ansicht]`



Abbildung A.42: Das View-Menü

Mit den Zoom-Funktionen läßt sich die Größe der Ansicht im aktiven Fenster verändern. **Zoom-Stufe zurücksetzen** stellt die Vergrößerung auf den Standard-Wert zurück, **Reinzoomen** bzw. **Rauszoomen** vergrößern bzw. verkleinern die gegenwärtige Ansicht. **Alles zeigen** setzt die Zoomstufe so, daß die ganze Domain in das aktive Fenster paßt.
Die derzeitige Zoomstufe wird unten links im Fenster angezeigt. Bei zu kleiner Zoomstufe wird das Gitter ausgeblendet.

Über **Point of interest** läßt sich die Ebene setzen, in der in den Fenstern Aufsicht, Vorderansicht und Seitenansicht gearbeitet wird. Im Fenster Aufsicht wird die Ebene (die "Tiefe", in der gearbeitet wird) durch den Z-Wert bestimmt, sie läßt sich über den Z-Wert bestimmen.

Entsprechendes gilt für für die Seitenansicht (X) und die Vorderansicht (Y).
Im Expertenmodus dieses Dialogfensters lassen sich neben diesen Werten für die Auf-, Vorder- und Seitenansicht die beiden übrigen Koordinaten setzen, die bestimmen, welcher Punkt in der Mitte des Fensters angezeigt wird.

Mit **Anzeigebereich** kann festgelegt werden, welche Objekte der Domain angezeigt werden sollen. Es lassen sich zwei Punkte (jeweils gegeben durch ihre drei Koordinaten) eingeben, die einen achsenparallelen Raum aufspannen. Alle Objekte, die vollständig in diesem Raum liegen, werden angezeigt. Im Experten-Modus läßt sich der sichtbare Bereich für jedes Fenster einzeln setzen.

Sichtbarkeit ermöglicht es, einzelne Boundaries auszublenden. In Aufsicht, Vorderansicht und Seitenansicht werden lediglich die Boundaries angezeigt, die in diesem Dialogfenster mit einem Häkchen versehen sind.

Es lassen sich beliebig viele dieser Dialogfenster öffnen, auf diese Weise kann über **Anwenden** zwischen den unterschiedlichen Einstellungen der verschiedenen Dialoge gewechselt werden.

Der Menüpunkt `[Optionen]`



Abbildung A.43: Das Options-Menü

Unter **Optionen** lassen sich etliche Einstellungen nach persönlichem Bedarf einstellen.
Domain-Einstellungen ermöglichen allgemeine Einstellungen wie den Gitterabstand, ob Nodes, Edges, Boundaries oder Finite Elemente (Quads, Tris, Hexas, Tetras) oder deren Nummern gezeichnet werden sollen oder ob Maus-Snapping benutzt werden soll. Maus-Snapping beschreibt die Toleranz einer Mauseingabe. Voreingestellt ist der Wert 10. Das bedeutet, daß für die Maus zwei Punkte, die im Abstand von weniger als zehn Punkten voneinander liegen, als ein Punkt behandelt werden, das bedeutet, daß insbesondere bei der Eingabe von Kanten, ein möglicherweise in der nähe liegender Punkt als der ausgewählte Punkt erkannt wird. Um das zu verhindern, drückt man bei der Eingabe die `SHIFT`-Taste. Man kann in diesem Menü Mouse-Snapping komplett ausschalten, oder für jedes Arbeitsfenster einzeln den Wert setzen.
Zusätzlich hat man die Möglichkeit, den Epsilon-Wert der Domain zu setzen. Dieser beschreibt, ab wann zwei Knoten innerhalb der Datenstruktur als gleich anerkannt werden. Das bedeutet, auch wenn Mousesnapping auf 0 steht und man zwei Punkte nebeneinander setzt, deren Ab-

stand den Epsilon-Wert überschreiten, wird intern nur ein einziger Punkt erzeugt und die von den zwei Punkten abhängigen Elemente hängen von nun an an dem gleichen Punkt.

Deswegen sollte man, *bevor* man mit der Arbeit an einer Domain beginnt, dem gewünschten Zoomlevel ensprechende Mouse-Snapping- und Epsilonwerte einstellen. Die voreingestellten Werte haben sich optimal für die Zoomstufe beim Starten des Programms (1.0x) erwiesen.

In **Farbeinstellungen** lassen sich die Farben für die einzelnen Objekte anpassen. Es lassen sich Farben für die ersten 16 Boundaries festlegen, existieren mehr als 16 Boundaries, so wiederholen sich die Farben zyklisch.

Farbtransparenz ermöglicht die Einstellung der Transparenz, mit der die Füllung von Boundaries, von 2D-Elementen (Tris und Quads) sowie von 3D-Elementen (Tetras und Hexas) versehen wird. Einzugeben sind ganze Zahlen zwischen 0 und 100, wobei 0 für vollkommen transparent und 100 für nicht transparent steht.

Um Einstellungen auf das 3D-Fenster anzuwenden, muß dieses zunächst aktualisiert werden (durch Anklicken des **Aktualisieren**-Knopfes).

Es lassen sich mehrere der Optionen-Dialoge öffnen, um schnell verschiedene Einstellungen aus den offenen Dialogen anwenden zu können. Mit **Zurücksetzen** werden sämtliche Einstellungen (also nicht nur die aus dem gerade geöffneten Dialog) auf Standard-Werte zurückgesetzt.

Die Einstellungen werden beim Beenden des Programms in einer Datei (`.gridoptions` auf Linux- und Unix-Systemen, `gridoptions.ini` auf Windows-Systemen) im persönlichen Verzeichnis des Benutzers gespeichert.

Der Menüpunkt `[Fenster]`



Abbildung A.44: Das Windows-Menü

Unter **Fenster** lassen sich die Aufsicht, die Vorder-, die Seiten- und die 3D-Ansicht maximieren, sowie alle Fenster auf die gleiche Größe setzen.

Zusätzlich lässt sich das 3D-Viewer-Fenster dazu bewegen, durch den Menüpunkt **Aktualisieren** die neuesten Änderungen in der Domain anzuzeigen. Dieser Menüpunkt ist der gleiche wie in der Werkzeugleiste **Projekt**, **Aktualisieren**.



Abbildung A.45: Das Help-Menü

Der Menüpunkt `[?]`

Im Hilfemenü läßt sich dieses Handbuch und der Aboutdialog, in dem alle Informationen über Grid und dessen Entwickler gegeben werden, aufrufen.

Laden und Speichern

Zum Speichern benutzt GRID das FEAST-Dateiformat, siehe Kapitel A.5.3.
Dieses Format hat gewisse Einschränkungen gegenüber der Domin, die intern in GRID verwendet wird, um die Daten zu halten, z.B. lassen sich die Daten nur in einem Modus speichern, also nur in 2D oder nur in 3D. Das Bedeutet, daß wenn z.B. in 2D gespeichert wird, die evtl. in der Domain enthaltenen 3D-Elemente(3D-Boundaries, Hexas und Tetras) nicht mitgespeichert werden. Umgekehrt gilt das nur für die Boundaries, da die anderen Elemente sehr wohl in 3D legale Elemente sind. Zusätzlich geht beim 2D-Modul die in der Domain implizit enthaltene dritte Koordinate beim Abspeichern verloren, das bedeutet, daß sie beim erneuten Laden auf 0 gesetzt wird. Zusätzlich werden Eigenschaften der Faces (Tris und Quads), wie Inner, Exter, Dirichlet oder Neumann nicht in 2D mitgespeichert, denn dort sind diese Flächen schon vollständige Macro-Objekte.
Auch beim Importieren gibt es einiges zu beachten. Die `obj.`-Dateien werden in eine Boundary Triangulation umgewandelt und auf dem Bildschirm dargestellt, wobei die Koordinaten der einzelnen Punkte nicht verändert werden. Daraus resultiert, daß, wenn ein Punkt der schon vorhandenen Domain und ein Punkt des neuen Objektes die gleichen Koordinaten bzgl. des eingestellten Epsilon-Wertes (s.o.) haben, sie als ein Punkt behandelt werden. Das bedeutet, daß kein neuer Punkt erzeugt wird, sondern der alte Punkt bekommt zusätzlich die Verweise auf die neuen Triangles.

> Ein Beispiel: Es wurde eine Kante mit den Punkten 1 und 2 eingefügt. Der Punkt 1 hat die Koordinaten: 0, 0, 0. Nun importiert man eine `obj.`-Datei, in der auch ein Punkt mit den gleichen Koordinaten wie Punkt 1 existiert. Es wird kein neuer Punkt erzeugt, sondern der Punkt 1 hat jetzt als Parent die Kante und ein BoundaryTriangulation-Objekt.

Im Moment wird das Java3D-Dateiformat eins zu eins interpretiert und in ein Boundary Triangulation-Objekt übersetzt. Allerdings enthält dieses Format einiges an Redundanz, so daß einige unerwünschte Dreiecke zusätzlich gezeichnet werden. In der nächsten GRID-Version wird dieses “Manko” beseitigt, indem man die unerwünschten Dreiecke herausfiltert.

A.5.4 Geometrien

In diesem Unterkapitel wird auf den allgemeinen Aufbau und Funktion der verwendeten Geometrien genauer eingegangen und beschrieben, welche Geometrie Untergeometrie von welcher ist und in welchem Modus die Geometrien nur auftreten dürfen.

Grundlegende Elemente

Node – Knoten Die Knoten werden durch ihre Koordinaten dargestellt. Es sind jeweils drei reelwertige[5] Koordinaten, x, y, und z. In 2D-Modus wird die dritte Koordinate ignoriert. Zusätzlich hat ein Knoten eine ganzzahlige Knotennummer, die für alle Knoten der Domain eindeutig ist und mit deren Hilfe die Knoten, z. B. in Dialogen, referenziert werden können.

Die Objekte basieren auf dem FEAST-Format und für das genauere Verständnis der Elemente wird angeraten dort die Feinheiten der Datenstruktur nachzulesen.

Die Knoten können, wie gesagt, in 3D- und 2D-Modus erzeugt werden und sie bilden die unterste Stufe in der Elementenhierarchie. Das bedeutet, daß sie kein Parent für andere Elemente bilden.

BoundaryNode und BoundaryNode3D Diese Art Knoten sind normale Knoten mit einer Knotennummer und drei Koordinaten. Diese Koordinaten sind nicht absolut, sondern werden bezüglich der Boundary gesetzt, der sie zugeordnet werden.

BoundaryNodes sind für den 2D-Modus bestimmt. Sie haben zusätzlich zu den Knoteneigenschaften einen Parameter und eine BoundaryNummer, die ein dem Knoten zugeordneten BoundaryObjekt referenziert. In 2D sind es BoundaryCircle und BoundarySegmentList. Der zweite Typ enthält eine Reihe von Segmenten und muß gesondert behandelt werden. Der Kreis besteht aus nur einem Segment, also kann mit dem Parameter die Position auf dem Kreis bestimmt werden. Die durch den Parameter definierte Zahl im Intervall [0, 1] beschreibt den prozentualen Anteil des Weges, den man geht, wenn man von der Drei-Uhr-Position auf dem Kreis startet und den Kreis gegen den Uhrzegersinn durchläuft. Genauso wird vorgegangen, wenn bei der Boundary Segment List nur ein Segment existiert. Wenn dieses Segment eine Kante(Edge) ist, dann wird der entsprechende Anteil beim Durchlauf vom Start- zum Endknoten bestimmt.

Wenn die Boundary Segment List mehrere Segmente hat, dann kann im Dialog das entsprechende Segment direkt angesprochen werden, oder, wenn man über die komplette Liste geht, dann gibt man als Parameter eine Dezimalzahl, bei der die Stellen vor dem Komma die Segmentnummer innerhalb der Liste bezeichnen und die Stellen nach dem Komma die eigentliche Position auf diesem Segment.

BoundaryNodes3D Diese Knoten werden relativ zu einem 3D-Boundary-Objekt gesetzt(Cubicle, Cylinder, Sphere und Triangulation). Sie haben zusätzlich zu den normalen Knoteneigenschaften noch zwei Winkel (Parameter), *alpha* und *beta* und die Nummer des zugeordneten Boundary-Objekt.

Die Positionierung geht intern folgendermaßen vor:

[5] Mit reelwertigen Koordinaten sind natürlich Werte vom Java-Typ `double` gemeint. Die genaue Beschreibung dieses Typs liefert [41].

1. Das Programm erstellt eine Kugel, die den Mittelpunkt in der Mitte des Boundary-Objekts hat und genau die Boundary beinhaltet.
2. Auf dieser Kugel wird mit Hilfe der zwei als Winkel übergebenen Parameter eine Position bezüglich des Punktes auf der Kugel auf 3 Uhr auf einer zur z-Achse senkrechten Ebene, die durch den Kugelmittelpunkt geht, bestimmt.
3. Ein Strahl wird durch den Kugelmittelpunkt und dem berechneten Punkt auf der Kugel gezogen.
4. Ein[6] Schnittpunkt mit dem Boundary-Objekt ist dann der gesuchte Punkt.

Edge – Kanten Die Kanten werden durch zwei Knoten repräsentiert. Sie sind in Grid gerichtet und haben einen Start- und einen Endknoten. Die Richtung spielt aber intern eine Rolle und hat für die Löser keine Relevanz. Die Kanten können in 2D- und 3D- Modus erzeugt werden und repräsentieren genauso wie die Knoten an sich noch keine vollständigen Geometrie-Objekte. Allerdings haben sie spezielle Eigenschaften, die für die Löser wichtig sind. Sie haben einen Kantenstatus, der entweder NONE, INNER oder EXTERIOR sein kann(siehe die Beschreibung des Moduls GRID, Kap. 3.5 oder die Beschreibung des FEAST-Formats, Kap. D.1). Zusätzlich haben sie noch die Boundary-Condition, die wiederum beschreibt, nach welchem Prinzip der Löser die Kanten verarbeitet (NEUMANN oder DIRICHLET, siehe dazu auch die Beschreibung des FEAST-Formats).

Die Tri-Objekte beschreiben Dreiecke im Raum und werden dementsprechend durch drei Knoten und drei Kanten dargestellt. In 2D sind das vollständige Geometrie-Objekte, die eine Teilmenge der Eigenschaften eines Macro-Objekts enthalten. Es handelt sich um den **Parallelblock**, der den Prozessor beschreibt, den der Löser verwenden soll, um dieses Objekt zu bearbeiten. Dadurch wird eine Lastverteilung ermöglicht. Der **Matrixblock** beschreibt den Teil der allgemeinen Gittermatrix, in dem sich das Element befindet. Der **Verfeinerungslevel** zeigt an, wie fein das Element in eine Gitterstruktur zerlegt wird.

Tri In 3D sind Tris Flächen, aus denen ein Tetra-Objekt (siehe unten) besteht. Hier haben sie keine Macro-Eigenschaften mehr, dafür aber Eigenschaften wie bei den Kanten: Flächenstatus (NONE/INNER/EXTERIOR) und die Boundary-Condition (DIRICHLET/NEUMANN).

Quad Die Quad-Objekte beschreiben Vierecke im Raum und werden dementsprechend durch vier Knoten und vier Kanten dargestellt. In 2D sind es vollständige Macro-Objekte und haben wie die Tris entsprechende Eigenschaften: **Parallelblock**, **Matrixblock** und den **Verfeinerungslevel**. Ein Anisotropie-Faktor kommt hinzu und zusätzlich existieren drei reelwertige Faktoren, die die Richtung der Anisotropie bestimmen. In 3D ist ein Quad eine Fläche eines Hexa-Macro-Objekts (siehe unten) und hat keine Macro-Eigenschaften, dafür aber die bei Tris schon erwähnten Eigenschaften: den Flächenstatus und die Boundary-Condition.

[6] Mit dem einen Schnittpunkt ist der gemeint, der vom Benutzer vorher in dem Dialog ausgewählt wurde. Eine Boundary Triangulation kann auch nicht konvex sein und der Strahl kann auch demenstprechend mehrere Schnittpunkte haben. Viele Löser akzeptieren nur konvexe 3D-Boundary-Objekte!

Tetra Die Tetra-Objekte beschreiben Tetraeder-Objekte, also Objekte, die aus vier Knoten, sechs Kanten und vier Tri-Objekten bestehen. Sie können nur im 3D-Modus erzeugt werden. Sie haben wie andere vollständige Geometrie-Objekte die Eigenschaften **Parallelblock**, **Matrixblock** und **Verfeinerungslevel**.

Hexa Die Hexa-Objekte bestehen aus 8 Knoten, 12 Kanten und 6 Quad-Objekten. Sie können genauso wie die Tetras nur im 3D-Modus erzeugt werden. Sie haben auch die Eigenschaften: **Matrixblock**, **Parallelblock**, **Verfeinerungslevel** und noch zusätlich ein Anisotropie-Faktor und 6 reelwertige Faktoren die wie bei 2D-Objekten die Richtung der Verfeinerung bestimmen.

Boundaries

Dieser Kapitel behandelt die möglichen Randbeschreibungen in GRID. Sie sind wiederum äquivalent mit den Randbeschreibungen des FEAST-Formats. Hier werden sie aus der Anwender-Sicht vorgestellt.

BoundaryCircle Das einfachste Randobjekt ist der Kreis. Er besteht aus einem Mittelpunkt und einem Radius. Der Kreis kann, wie schon oben erwähnt, allein, oder als Kreissegment in einer BoundarySegmentList (siehe unten) vorkommen. Deswegen ist es prinzipiell möglich anhand von zwei Winkeln, wiederum ausgehend von der Drei-Uhr-Position, auf dem Kreis das Segment anzugeben.

Die Drei-Uhr-Position bestimmt den Winkel 0, von dort ausgehend werden die Winkel gegen den Uhrzeigersinn größer.

Ein Kreis kann nur im 2D-Modus vorkommen.

BoundarySegmentList Eine BoundarySegmentList besteht aus sogenannten Segmenten. Diese können Kreissegmente oder Kanten sein. Sie bilden immer ein geschlossenes Polygon. SegmentListen können nur im 2D-Modus erstellt werden. Sie brauchen nicht konvex zu sein.

BoundaryCubicle / BoundaryCube Ein BoundaryCubicle-Objekt ist so etwas wie ein Boundary-Hexa-Objekt im 3D-Modus. Es besteht auch aus 8 Knoten. Allerdings muß es konvex sein.

BoundarySphere Ein 3D-Äquivalent zu den Kreisen in 2D. Allerdings können keine Kugelsegmente angegeben werden, da es kein Äquivalent zu den BoundarySegmentListen existiert. Eine Kugel besteht aus einem Mittelpunkt und Radius.

BoundaryCylinder Ein Zylinder-Randobjekt beschrieben durch zwei Punkte, Top und Bottom, die die Mittelachse des Zylinders bilden. Zusätzlich beschreibt der Radius die Dicke des Zylinders im Raum. Ein Zylinder ist schon per Definition ein dreidimensionales Objekt und kann deswegen nur im 3D-Modus verwendet werden.

Boundary Triangulation Dies ist das einzige Boundary-Objekt, das in GRID nicht direkt bearbeitet werden kann. Man kann aber eine `obj.`-Datei importieren, die automatisch als ein Boundary Triangulation-Objekt geladen wird, und diese in die Domain einbinden (skalieren, drehen, etc.) Es besteht intern aus Punkten, die durch Dreiecke verbunden werden. Dieses Gebilde sollte für die Löser konvex sein.

A.5.5 Ein 2D-Beispielproblem

In diesem Kapitel wird beispielhaft erläutert, wie sich die Geometrie für den DFG-Benchmark *Flow around a circle* anlegen lässt. Mehr Details über dieses Beispiel stehen in *The Virtual Album of Fluid Motion* unter `http://www.featflow.de/album/` bereit.
Die Geometrie für dieses Beispiel ist recht einfach aufgebaut: Sie besteht aus der äußeren Boundary, einem achsenparallelen Rechteck, sowie einer inneren Boundary, einem Circle im linken Bereich des Rechtecks.

1. GRID starten.

2. In den 2D-Modus wechseln. Da im 2D-Modus gearbeitet wird, kann die Z-Koordinate ignoriert werden.

3. In den Optionen den Gitterabstand auf 1 stellen, das Maus-Snapping einschalten, den Snapping-Abstand für X und Y auf 1 stellen.

4. Den Anzeigebereich auf $X = -5, Y = -5, X = 10, Y = 5$ einstellen.

5. In das Fenster zoomen, so daß der Anzeigebereich ausreichend groß dargestellt wird. Den Bereich in die Mitte des Fensters schieben.

6. Das Segment-List-Werkzeug auswählen.

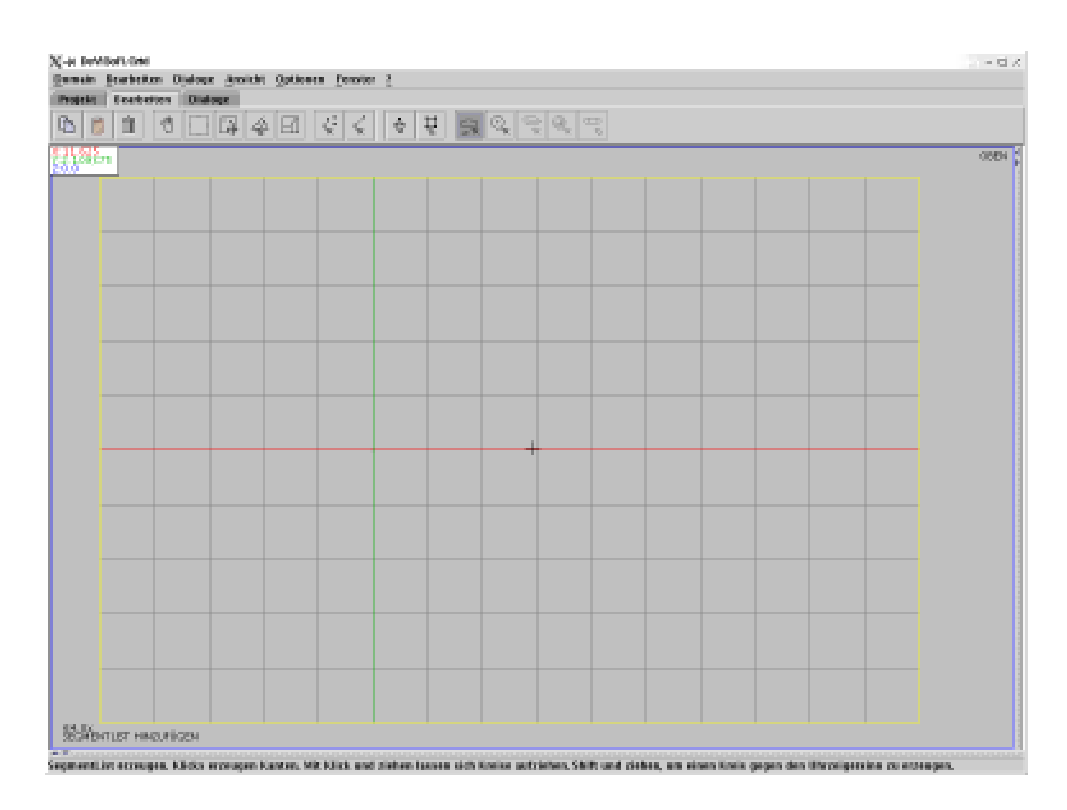


Abbildung A.46: 2D – Anzeigebereich vorbereiten

7. Gegen den Uhrzeigersinn die Eckpunkte $(-4, 4)$, $(-4, -4)$,$(4, -9)$, $(9, 4)$ sowie wieder den ersten Eckpunkt anklicken. Damit wird die SegmentList angelegt.

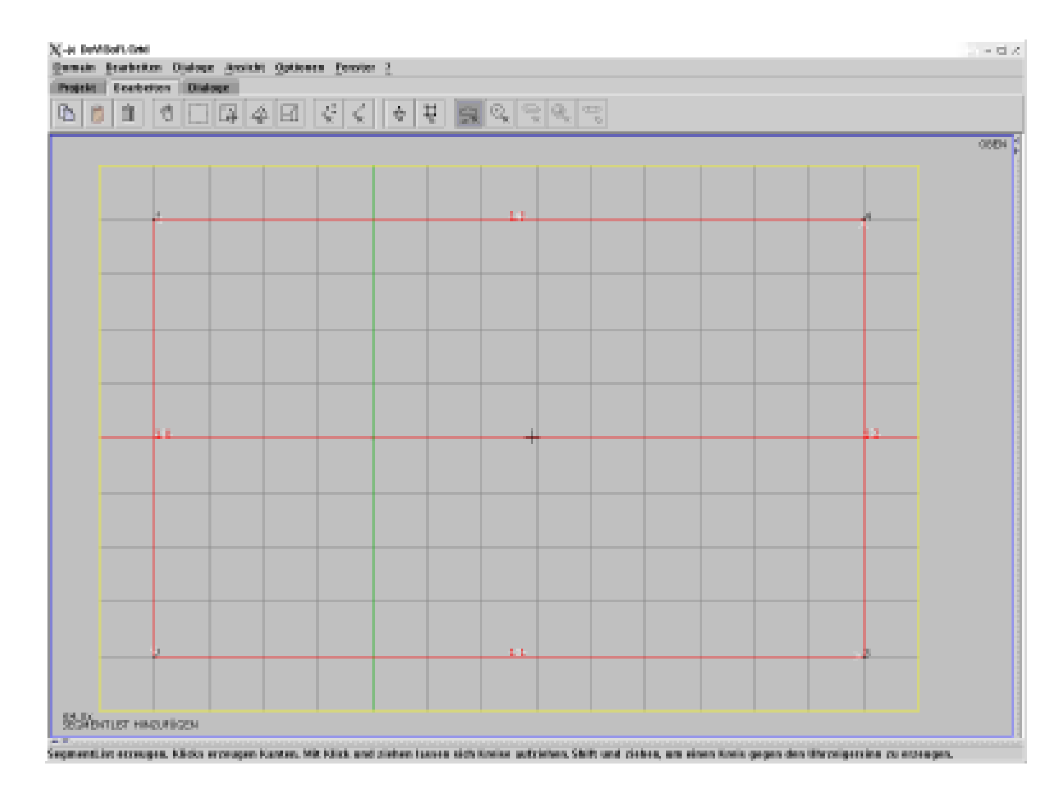


Abbildung A.47: 2D – SegmentList anlegen.

8. Da der Editor noch keine Undo-Funktion unterstützt, sicherheitshalber speichern.

9. Einen Circle mit Mittelpunkt $(0, 0)$ und Radius 3 anlegen. Dazu

   - entweder das Maus-Werkzeug Circle
   - oder den Circle-Dialog wählen. Um den Circle-Dialog zu benutzen, muß der Mittelpunkt an der Stelle $(0, 0)$ bereits existieren.

   Als innere Boundary muß der Circle im Uhrzeigersinn gerichtet sein, deshalb ist im Circle-Dialog ein Startwinkel von 360 Grad und ein End-Winkel von 0 Grad anzugeben.

10. Den Rechenbereich mit Quads füllen.

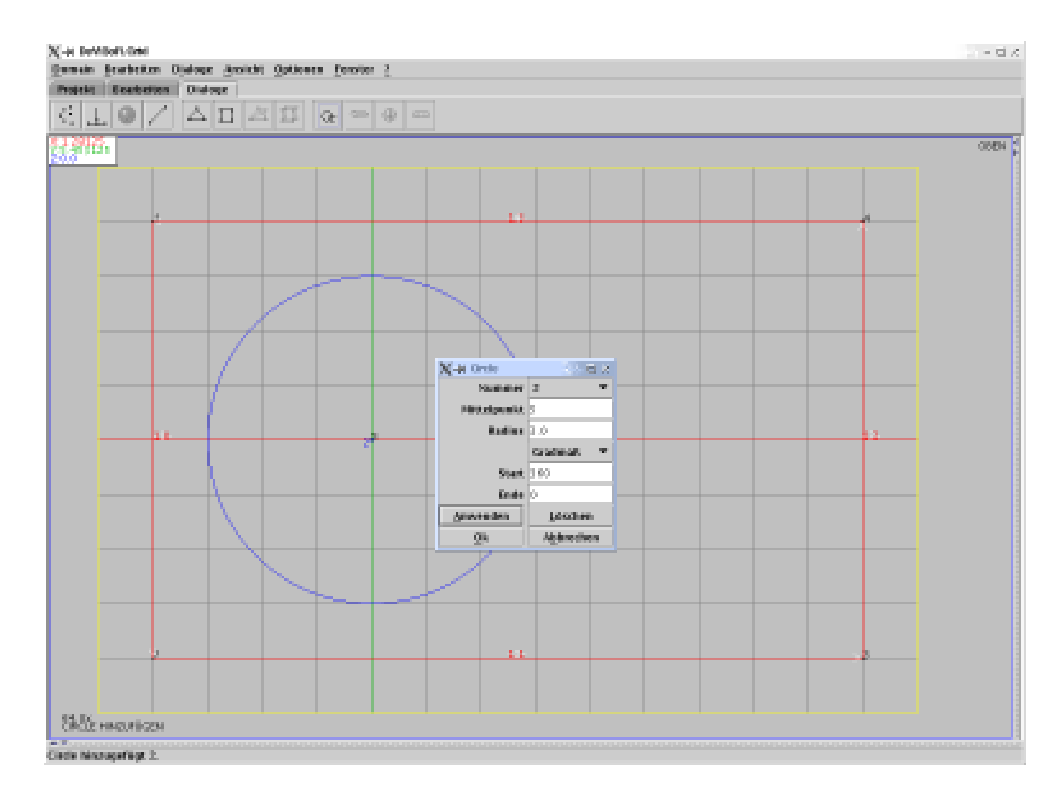


Abbildung A.48: 2D – Circle malen.

A.5.6 Ein 3D-Beispielproblem

Das dreidimensionale Beispielproblem sieht ähnlich aus wie das unter beschriebene Problem. Es besteht aus einem achsenparallelen Cubicle als äußerem Boundary und einer Sphere als innerem.

1. GRID starten.

2. In den 3D-Modus wechseln.

3. In den Optionen den Gitterabstand auf 1 stellen, das Maus-Snapping einschalten, den Snapping-Abstand für X, Y und Z auf 1 stellen.

4. Den Anzeigebereich auf $X = -5, Y = -5, Z = -5, X = 10, Y = 5, Z = 5$ einstellen.

5. In die Fenster zoomen, so daß der Anzeigebereich ausreichend groß dargestellt wird. Den Bereich in die Mitte des Fensters schieben.

6. Damit das 3D-Fenster stets aktualisiert wird, die Funktion zum automatischen Update auswählen.

7. Das Cubicle-Werkzeug auswählen.

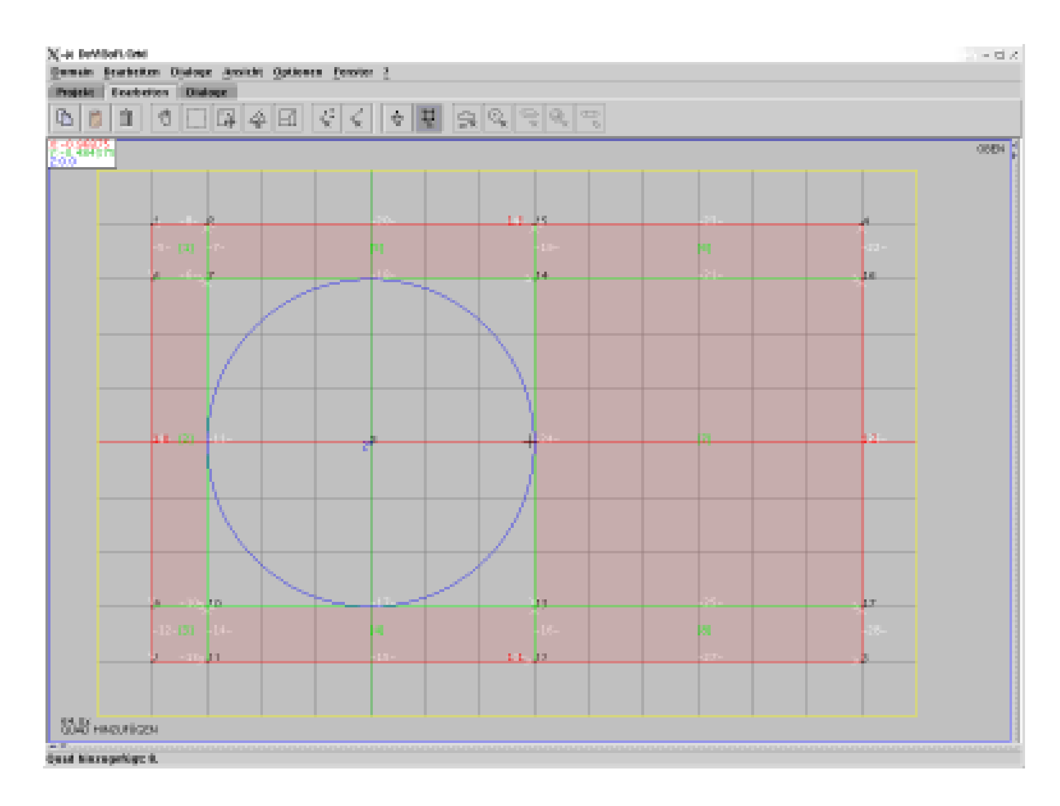


Abbildung A.49: 2D – Rechenbereich mit Quads füllen

8. Mit dem Point-of-interest-Dialog oder der rechten Maustaste im Aufsicht-Fenster in die Ebene $Z = -4$ wechseln.
9. Den ersten Punkt $-4, 4, -4$ anlegen.
10. Mit dem Point-of-interest-Dialog oder der rechten Maustaste im Aufsicht-Fenster in die Ebene $Z = 4$ wechseln.
11. Den zweiten Punkt $9, -4, -4$ anlegen. Der Cubicle-Modus erwartet, daß die Rotation angegeben wird. Da das Cublicle nicht gedreht werden soll, noch ein weiteres Mal an diese Stelle klicken, das Cubicle wird nun erzeugt.
12. Da der Editor noch keine Undo-Funktion unterstützt, sicherheitshalber speichern.
13. Einen Sphere mit Mittelpunkt $(0, 0, 0)$ und Radius 2 anlegen. Dazu
    - entweder das Maus-Werkzeug Sphere
    - oder den Sphere-Dialog wählen.
14. Jetzt muß der Rechenbereich mit Hexas gefüllt werden. Dazu werden ebenenweise Quads definiert, die anschließend zu Hexas ergänzt werden.

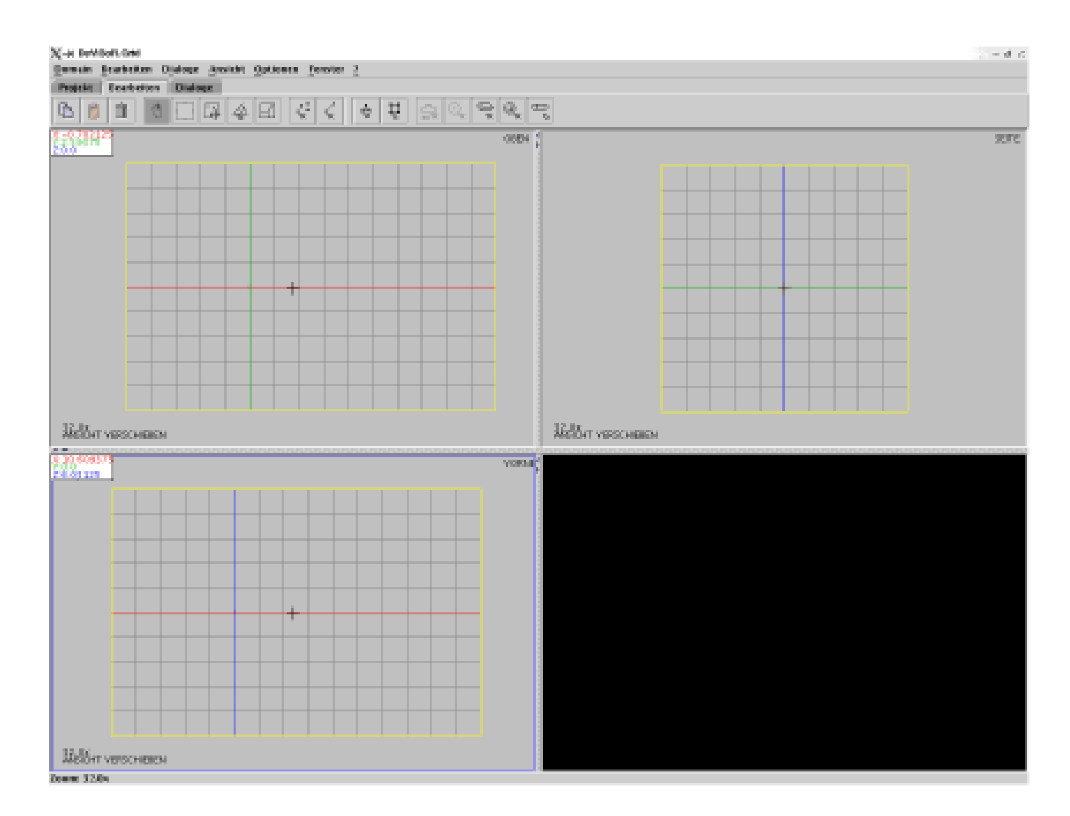


Abbildung A.50: 3D – Anzeigebereich vorbereiten

- Nacheinander die Ebenen $Z = -4$, $Z = -2$, $Z = 2$ und $Z = 4$ auswählen. Damit in den Ebenen stets neue Punkte angelegt werden und die Eckpunkte der Quads nicht aus den bereits bestehenden Punkten in anderen Ebenen gewählt werden, die Umschalttaste beim Zusammenklicken der Quads gedrückt halten.

  Bei Bedarf kann auch der Anzeigebereich auf die jeweilige Ebene eingeschränkt werden.

- Damit die Quads Hexas bilden können, mußbestimmt werden, wie sie zueinander liegen. Das passiert mit Edges. Für die ersten Edges in der Seitenansicht die X-Ebene auf -2 einstellen und die Eckpunkte der Quads, die einen Hexa bilden werden, miteinander verbinden. Das gleiche für die Ebene $X = 2$.

- Anschließend im Hexa-Dialog die Hexas aus den passenden Quads und Edges zusammensetzen.

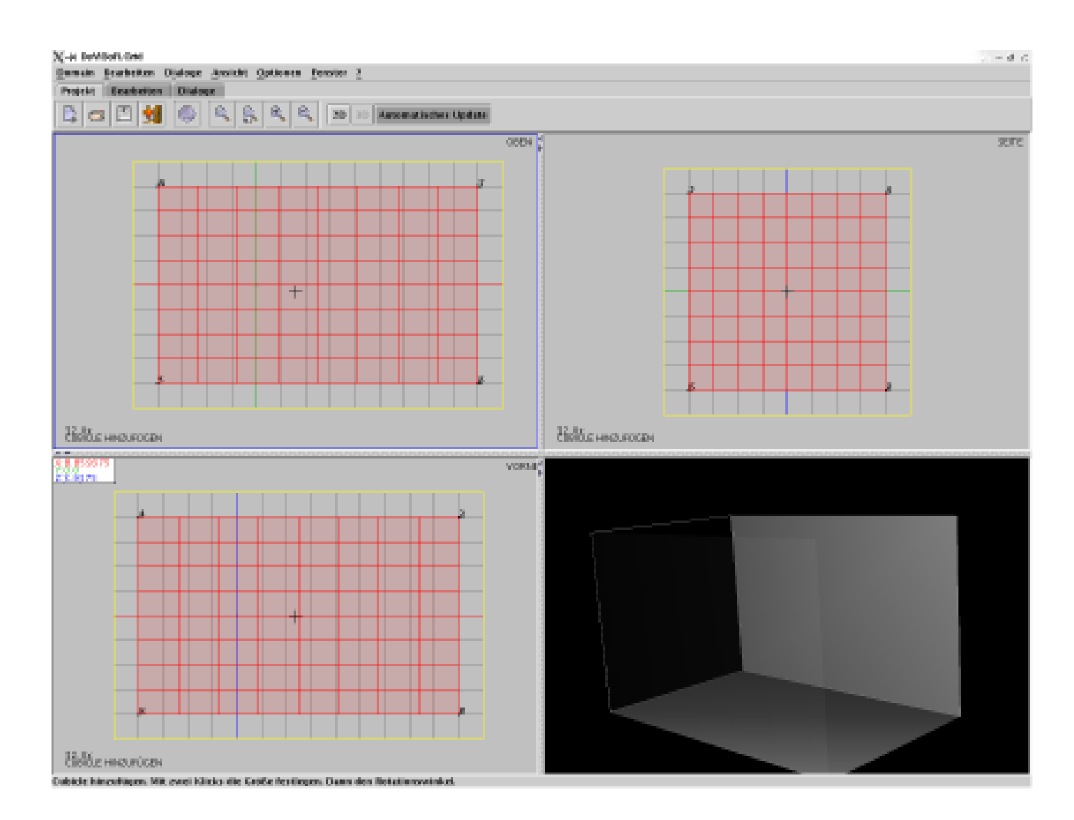


Abbildung A.51: 3D – Cubicle malen

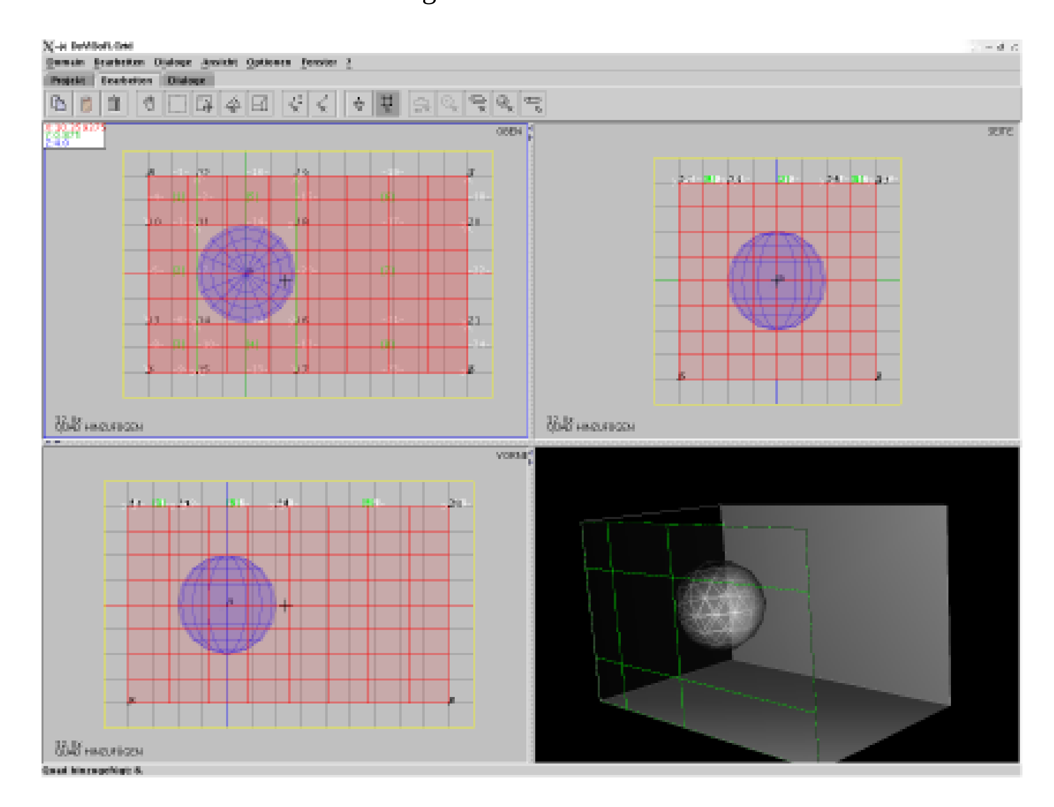


Abbildung A.52: 3D – Sphere zeichnen

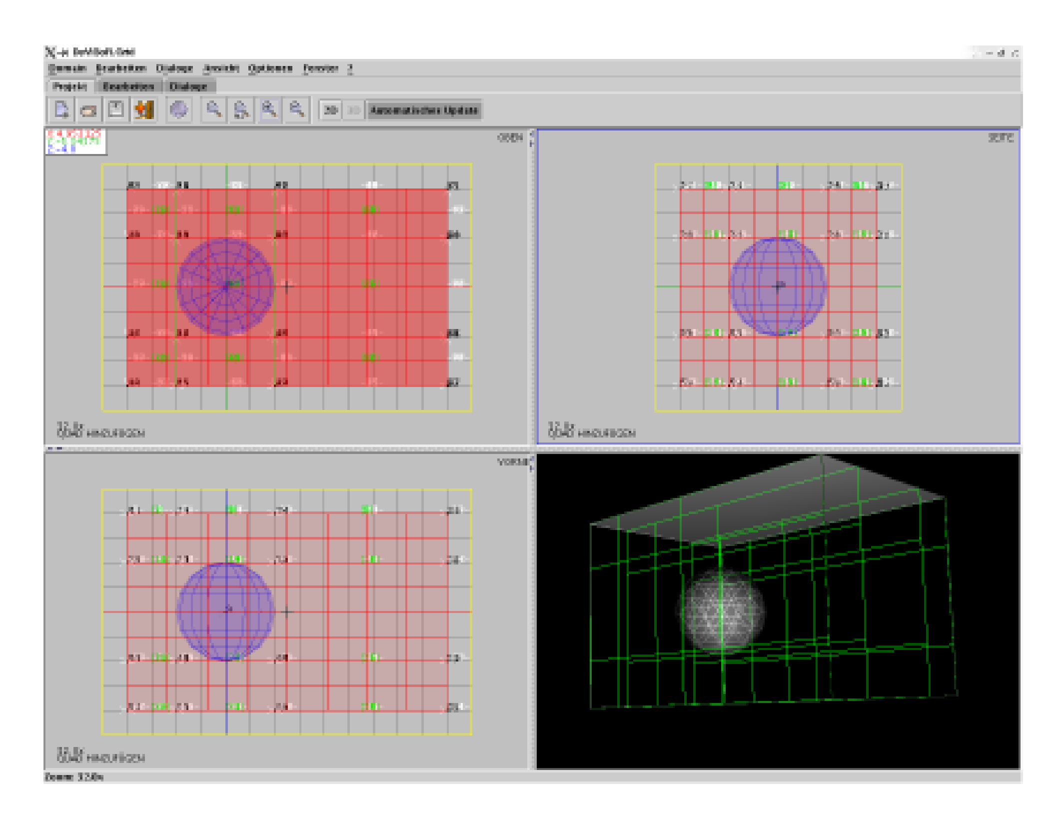


Abbildung A.53: 3D – Rechenbereich mit Quads füllen

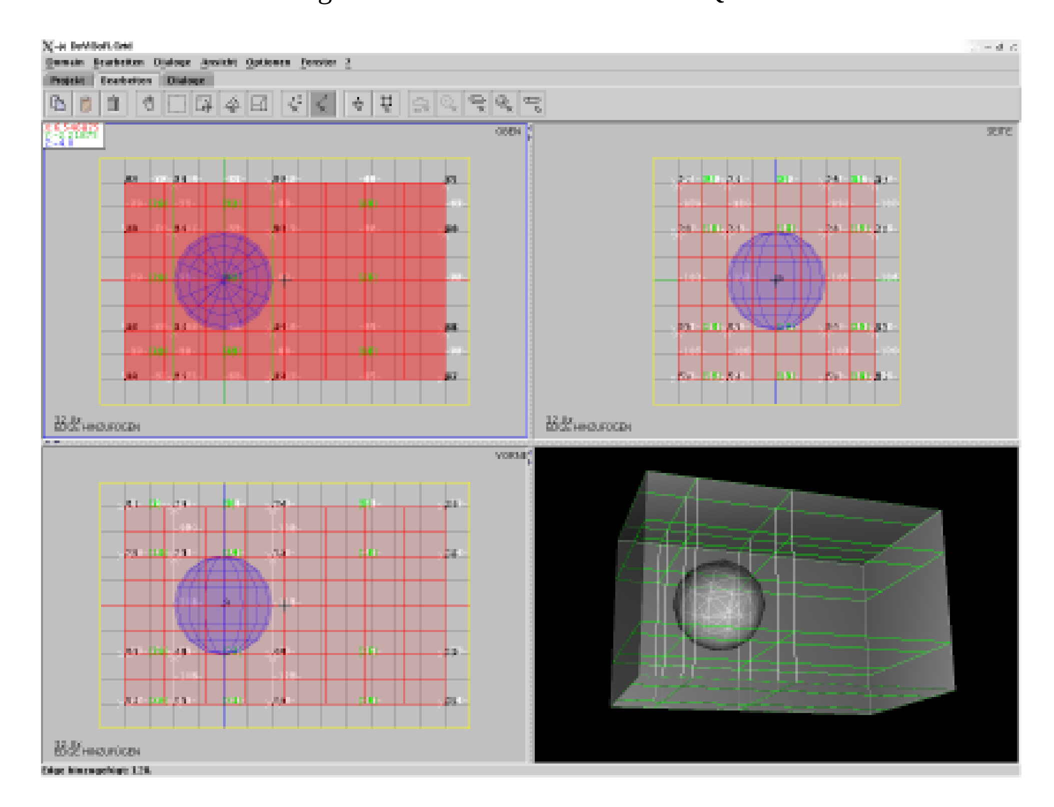


Abbildung A.54: 3D – Aus Quads werden Hexas 1

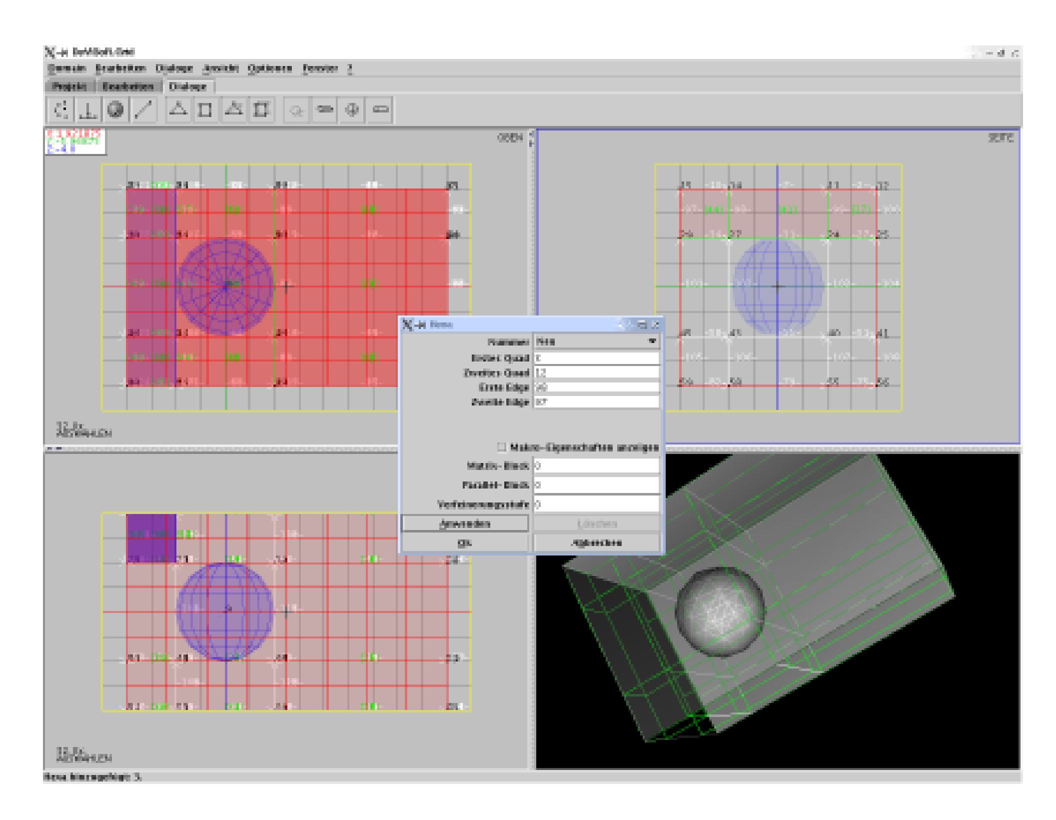


Abbildung A.55: 3D – Aus Quads werden Hexas 2

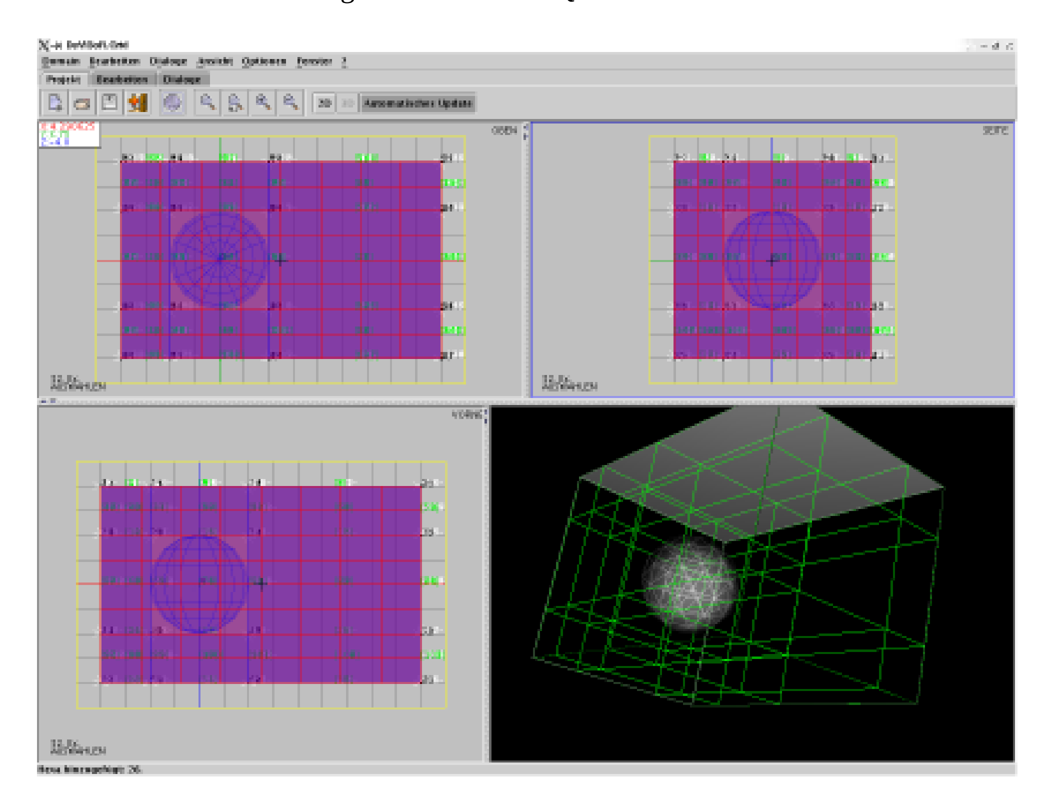


Abbildung A.56: 3D – Aus Quads werden Hexas 3

A.6 VISION — Die Visualisierung der Ergebnisse

A.6.1 Einleitung

Da VISION ein Teil des DEVISOR-Pakets ist, ist es möglich, die Pipe vom Kontrollmodul aufzurufen und die Ergebnisse direkt zur weiteren Bearbeitung weiterzuvermitteln.

A.6.2 Die Komponenten der Pipe

Filter

CutlinesFilter Der CutlinesFilter benötigt als Parameter die Angabe der Schnittlinie. Diese wird mit Hilfe der folgenden Parameter angegeben.

- **start point of cutline**
  Dieser Parameter gibt einen Punkt in der Ebene an, durch den die Cutline läuft.
- **normal vector of cutline**
  Gibt den Normalenvektor der Cutline an.

CutsurfacesFilter Der CutsurfacesFilter benötigt als Parameter die Angabe der Schnittebene. Diese wird mit Hilfe der folgenden Parameter angegeben.

- **start point of cutplane**
  Dieser Parameter gibt einen Punkt im Raum an, durch den die Cutplane läuft.
- **normal vector of cutplane**
  Gibt den Normalenvektor der Cutplane an.

IdentityFilter Dieser Filter gibt die Domain vollkommen ungefiltert zurück und benötigt daher keine gesonderten Parameter.

IsoLinesFilter Der IsoLinesFilter benötigt für eine korrekte Filterungsfunktion eine beliebige Anzahl von Zahlenwerten, um für diese jeweils eine Isolinie entlang der Punkte mit den entsprechenden Werten in der Domain generieren zu können.

- **iso values**
  Es können beliebig viele Isowerte angegeben werden, die jeweils zur Generierung einer Isolinie führen.

ScalarThresholdFilter Dieser Filter schneidet die Werte innerhalb der Domain auf einen vorgegebenen Bereich zu. Dazu werden die folgenden Parameter benötigt:

- **max value**
  Der Wert, oberhalb dessen alle Werte auf den gegebenen Maximalwert gesetzt werden.
- **min value**
  Der Wert, unterhalb dessen alle Werte auf den gegebenen Minimalwert gesetzt werden.
- **variable name**
  Der Name der Variablen, die gefiltert werden soll (z.B. ”pressure”).

Surface3DFilter Dieser Filter benötigt für eine korrekte 3D-Darstellung einer 2D-Domain einen Skalierungsfaktor, um die Z-Werte geeignet darstellen zu können.

- **scale parameter**
  Der Skalierungsfaktor, mit dem die Z-Werte auf eine geeignete Skala skaliert werden können.

Mapper

Arrow2DMapper Der Arrow2DMapper visualisiert Vektoren in einer 2D-Domain durch gerichtete Pfeile. Die Parameter:

- **Visualization Method**
  Beeinflusst die Art, in der die Werte dargestellt werden: Einstellbar sind Visualisierung durch a) Dicke der Pfeile, b) Länge der Pfeile, und c) Farbe der Pfeile.

- **Maximum Arrow Length**
  Legt die maximale Länge der Pfeile im erzeugten Scene-Graphen fest.

- **Arrowlength**
  Legt die konstante Länge der Pfeile bei ausgewählter Visualisierungsmethode ”Visualisierung durch Farbe” fest.

- **Arrowwidth**
  Legt die konstante Dicke der Pfeile fest.

- **Arrowcolor**
  Legt die konstante Farbe der Pfeile bei ausgewählter Visualisierungsmethode ”Visualisierung durch Länge/Dicke” fest.

- **MaxColorValue**
  Bestimmt die obere Schranke des Farbraums auf der verwendeten Legende.

- **MinColorValue**
  Bestimmt die untere Schranke des Farbraums auf der verwendeten Legende.

Arrow3DMapper Der Arrow3DMapper visualisiert Vektoren in einer 3D-Domain durch gerichtete Pfeile. Die Parameter:

- **Visualization Method**
  Beeinflusst die Art, in der die Werte dargestellt werden: Einstellbar sind Visualisierung durch a) Dicke der Pfeile, b) Länge der Pfeile, und c) Farbe der Pfeile.

- **Maximum Arrow Length**
  Legt die maximale Länge der Pfeile im erzeugten Scene-Graphen fest.

- **Arrowlength**
  Legt die konstante Länge der Pfeile bei ausgewählter Visualisierungsmethode ”Visualisierung durch Farbe” fest.

- **Arrowwidth**
  Legt die konstante Dicke der Pfeile fest.

- **Arrowcolor**
  Legt die konstante Farbe der Pfeile bei ausgewählter Visualisierungsmethode ”Visualisierung durch Länge/Dicke” fest.

- **MaxColorValue**
  Bestimmt die obere Schranke des Farbraums auf der verwendeten Legende.

- **MinColorValue**
  Bestimmt die untere Schranke des Farbraums auf der verwendeten Legende.

DomainFineGridMapper Arbeitet wie der DomainMapper, mit der Erweiterung der Visualisierung der verfeinerten Zellen. Als Parameter erforderlich ist zusätzlich ein Vektor mit den Indizes der Zellen, welche verfeinert werden sollen.

DomainMapper Der DomainMapper stellt eine Möglichkeit zur Visualisierung eines Gitters in 3D dar.

- **QuadColor**
  Gibt an, in welcher Farbe die Kanten, deren Parent-Element ein Quad ist, gezeichnet werden sollen. Einstellbar ist diese Farbe als Kombination aus Rot, Grün und Blau-Wert im Bereich von 0 bis 255.

- **HexColor**
  Gibt an, in welcher Farbe die Kanten, deren Parent-Element ein Hex ist, gezeichnet werden sollen. Einstellbar ist diese Farbe als Kombination aus Rot, Grün und Blau-Wert im Bereich von 0 bis 255.

- **TetraColor**
  Gibt an, in welcher Farbe die Kanten, deren Parent-Element ein Tetra ist, gezeichnet werden sollen. Einstellbar ist diese Farbe als Kombination aus Rot, Grün und Blau-Wert im Bereich von 0 bis 255.

- **TriColor**
  Gibt an, in welcher Farbe die Kanten, deren Parent-Element ein Tri ist, gezeichnet werden sollen. Einstellbar ist diese Farbe als Kombination aus Rot, Grün und Blau-Wert im Bereich von 0 bis 255.

- **Show Boundaries as a grid**
  Gibt an, ob die in der Domain enthaltenen Boundary-Objekte als ein semi-transparentes Gitter gezeichnet werden sollen, oder nicht.

- **Show Boundaries as a filled shape**
  Gibt an, ob Boundary-Objekte als semi-transparente Fläche gezeichnet werden soll.

- **Boundary Transparency**
  Gibt den Transparenz-Wert fuer die Boundary-Objekte an. Gültige Werte liegen im Bereich zwischen 0 (nicht transparent) und 1 (vollständig transparent).

NullMapper Der NullMapper stellt die Domain nicht dar, sondern liefert eine leere Domain zurück an den Renderer. Dieser Mapper hat somit keine Parameter.

ParticleTracingMapper Der ParticleTracingMapper bietet auf der obersten Ebene folgende Parameter als Einstellungsmöglichkeiten an

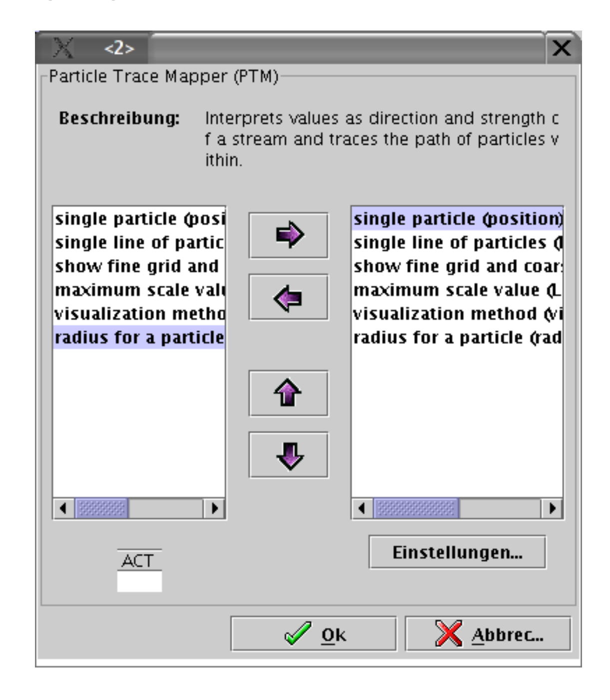


Abbildung A.57: Die Parameter für Particle Tracing

- **single particle**
  Ermöglicht einzelne Partikel, durch Koordinatenangabe, in das Gebiet zu plazieren. Die Eingabe der Koordinaten für Partikel ist immer in 3D. Ist das zu visualisierende Gebiet ein 2D Gebiet, wird die dritte Koordinate nicht beachtet.
- **single line of particles** Dieser Parameter ermöglicht es, eine Linie zu definieren, auf welcher in äquidistanten Abständen Partikel gesetzt werden. Um diese Linie zu definieren, sind zwei Punkte notwendig. Diese werden wie bei den **single Particle**-Parameter durch Koordinaten eingegeben. Die Abstände zwischen den Partikeln auf einer Linie werden parametrisch angegeben.
- **show fine grid and coarse grid** Hier kann man angeben, ob man zusätzlich zum Grobgitter auch das Feingitter angezeigt werden soll.
- **maximum scale value** Dieser Wert gibt den maximalen Wert der Legende an. Um ideale Visualisierungen zu ermöglichen, muß dieser Wert an die Ergebnisse angepaßt sein. Werden die Werte nicht abgestimmt, kann es sein, daß das Intervall der Farblegende zu groß oder zu klein ist, um Unterschiede richtig darzustellen.
- **visualisation method** Hier kann man aus den drei angebotenen Methoden eine Auswählen.

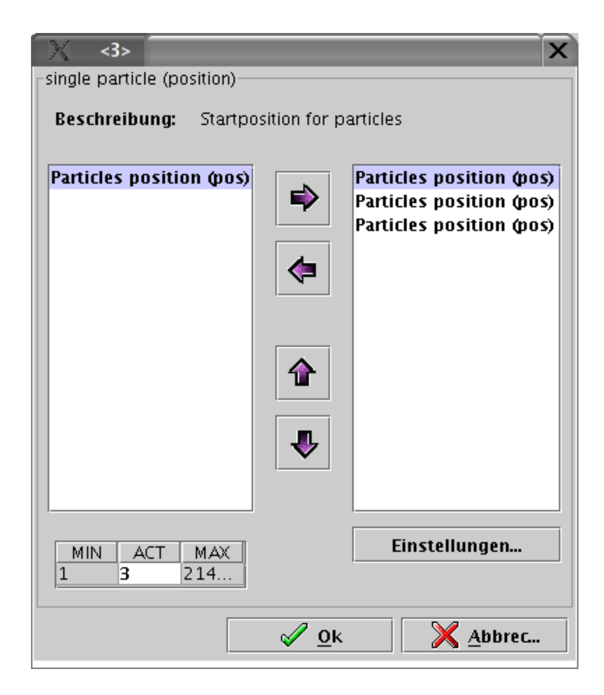


Abbildung A.58: Einzelne Partikel erstellen

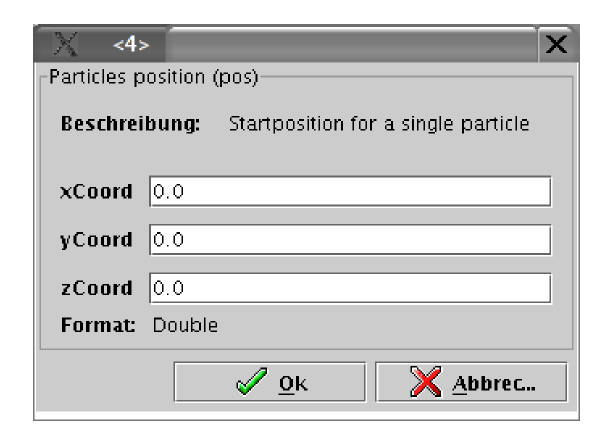


Abbildung A.59: Koordinaten für einzelne Partikel

- **radius for particle** Um die bestmögliche Visualisierung zu bekommen, kann hier der Radius der Partikel gesetzt werden.

ShadedPlotMapper Der ShadedPlotMapper hat zur korrekten Darstellung einer Domain folgende Konfigurationsmöglichkeiten.

- **legend style**
  Gibt den Stil an, in dem die Legende dargestellt werden soll. Die Legende kann für verschiedene Zwecke konfiguriert werden, z.B. zur Anzeige von Temperaturen.

- **show edges**
  Gibt an, ob die Kanten der Domain angezeigt werden sollen.

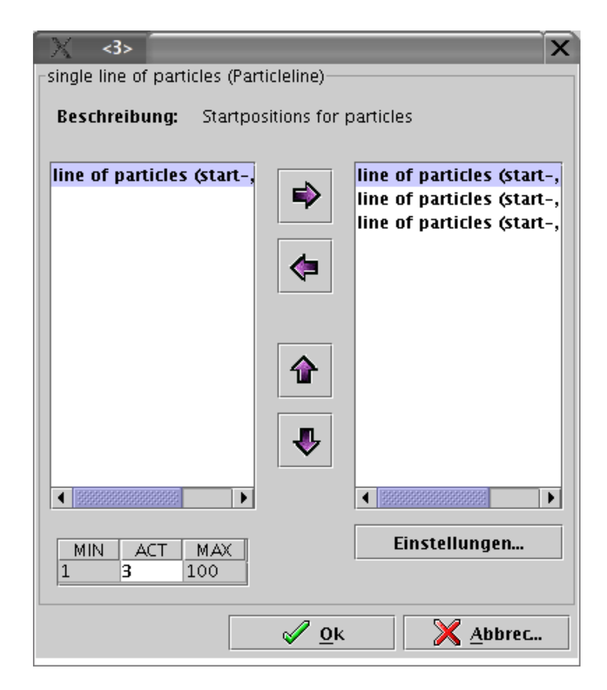


Abbildung A.60: Erstellen einer Partikellinie

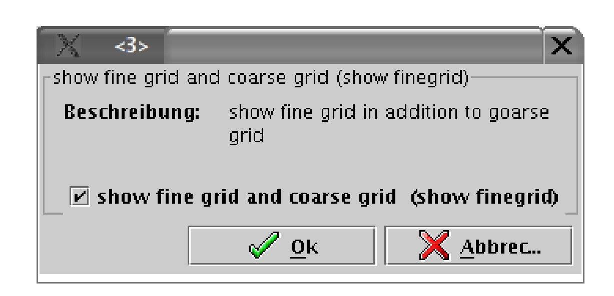


Abbildung A.61: Anzeigen des Feingitters

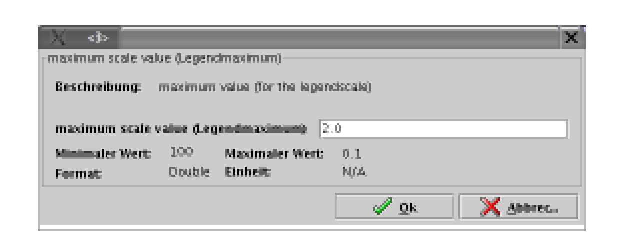


Abbildung A.62: Maximalwert der Legende

- **show nodes**
  Gibt an, ob die Knoten der Domain angezeigt werden sollen.

- **show elements**
  Gibt an, ob die Flächen der Domain angezeigt werden sollen.

- **variable name**

Gibt den Variablennamen an, dessen Werte Grundlage des ”shaded plot” werden sollen (z.B. ”pressure”).

- **max value**
  Der maximal anzeigbare Wert innerhalb der Domain. Größere Werte werden schwarz dargestellt. Dieser Parameter ist insbesondere wichtig für die Darstellung der Legende.

- **min value**
  Der minimal anzeigbare Wert innerhalb der Domain. Kleinere Werte werden schwarz dargestellt. Dieser Parameter ist insbesondere wichtig für die Darstellung der Legende.

Sphere2DMapper Stellt skalare Werte an den Knoten zweidimensionaler Domains durch Kreise dar. Die Parameter, die sich von denen bei den ArrowMappern unterscheiden, sind im einzelnen:

- **Scalar Value Name**
  Der Name des Wertes in der erzeugten GMV-Datei, der visualisiert werden soll.

- **Visualiszation Method**
  Die Art und Weise, auf der die Werte dargestellt werden sollen. Möglich sind hier: Radius und Farbe der Kreise.

- **SphereColor**
  Legt die Farbe der Kreise bei Visualisierungsmethode ”Radius” fest.

- **SphereRadius**
  Legt den Radius der Kreise bei Visualisierungsmethode ”Farbe” fest.

Sphere3DMapper Stellt skalare Werte an den Knoten dreidimensionaler Domains durch Kugeln dar. Die Parameter, die sich von denen bei den ArrowMappern unterscheiden, sind im einzelnen:

- **Scalar Value Name**
  Der Name des Wertes in der erzeugten GMV-Datei, der visualisiert werden soll.

- **Visualiszation Method**
  Die Art und Weise, auf der die Werte dargestellt werden sollen. Möglich sind hier: Radius und Farbe der Kugeln.

- **SphereColor**
  Legt die Farbe der Kugeln bei Visualisierungsmethode ”Radius” fest.

- **SphereRadius**
  Legt den Radius der Kugeln bei Visualisierungsmethode ”Farbe” fest.

Renderer

SimpleRenderer Der SimpleRenderer kapselt die Darstellung der Domain, er benötigt aber keine Parameter und hat daher auch keine Benutzerschnittstelle.

Das VISION-Fenster

Das VISION-Fenster zeigt die Domain in der konfigurierten Visualisierungstechnik in einem Fenster mitsamt Legende an. Die Bedienung des VISION-Fensters wird bereits in Kapitel A.5.3 beschrieben. Durch einen Klick auf “Ok” wird die nächste Berechnung in der Pipe durchgeführt und anschließend angezeigt, falls die Anzeige des nächsten Zeitschrittes konfigurativ erwünscht war.

Anhang B

Erweiterungsmöglichkeiten

B.1 Einbau weiterer Visualisierungstechniken in das VISION-Modul

In diesem Abschnitt wird mittels kleiner Beispiele erläutert, welche Arbeitsschritte notwendig sind, um neue Visualisierungen zu integrieren. Da der Datenfluß innerhalb des VISION-Pakets durch eine Visualisierungs-Pipeline läuft, der in drei Teile gegliedert ist (vgl. Kap. 3.6.8), folgt auch dieses Kapitel dieser Gliederung, um den Besonderheiten jedes Abschnitts der Pipeline gerecht zu werden.
Die zentrale Datenstruktur, die **VisionDomain**, wird in Kap. 3.6.9 beschrieben.

B.1.1 Filter

Um einen Filter in die Pipeline zu integrieren, sind vier einfache Arbeitsschritte notwendig:

1. Das Schreiben einer Klasse, die das Interface **devisor.vision.pipe.filter.I_Filter** und das Interface **devisor.vision.I_Configurable** implementiert. Das passiert typischerweise, indem sie von der Klasse **AbstractFilter** aus demselben Paket erbt.

2. Die Implementierung der fehlenden Methoden.

3. Das Einfügen des neuen Pipeline-Elements zu der XML-Datei **devisor.vision.util.Configurables.xml**

4. Und zuletzt das Hinzufügen einiger Internationalisierungs-Strings zu der Standard-Datei **devisor.vision.util.FindConfigurablesi18n.properties**, sowie u.U. auch zu den anderen Internationalisierungs-Dateien in diesem Paket.

Die Durchführung dieser vier Schritte ist Gegenstand der nächsten vier Abschnitte.
Der Filter, der gebaut wird, ist der **Identity**-Filter, der auch schon so im VISION-Paket enthalten ist. Dieser Filter verändert die Daten nicht, ist extrem einfach zu implementieren und daher gut geeignet, die notwendigen Schritte zu erklären.

Die neue Filter-Klasse

Zunächst wird eine Klasse benötigt, die von **AbstractFilter** erbt. In Codebeispiel 10 ist zu sehen, wie so eine Klasse im einfachsten Falle aussehen könnte.

```
public class IdentityFilter extends AbstractFilter {

    public IdentityFilter(){
        super();
    }

    public boolean filter(){
    }

    public ParameterList getParameterList() {
    }

    public ListOfErrors configure(ParameterList aList){
    }

}
```

Codebeispiel 10: Grundgerüst jeder Filterklasse

Die fehlenden Methoden

Natürlich ist diese Klasse so noch nicht kompilierbar, weil die geerbten Methoden noch zu implementieren sind:

public ParameterList getParameterList()
Diese Methode soll alle Parameter zur Verfügung stellen, die von diesem Filter benötigt werden. Sie wird bei der Initialisierung des VISION-Moduls aufgerufen, um alle Parameter aller Elemente der Pipeline zu sammeln und komplett an das CONTROL-Modul zu schicken, damit sie dort dem Benutzer zur Verfügung gestellt werden können.

Parameterlisten enthalten nicht nur den Namen und u.U. einen Default-Wert, sondern auch eine Beschreibung der GUI-Repräsentation dieses Parameters, Internationalisierungs-Angaben und noch einiges mehr.

Der Aufbau einer Parameterliste kann daher sehr komplex werden, aber das Beispiel kommt mit folgenden Zeilen aus:

```
    public ParameterList getParameterList() {
        return EmptyParameterList.create();
    }
```

Codebeispiel 11: Die einfachste gültige Parameterliste

Um die Daten *nicht* zu verändern, braucht man selbstverständlich auch keine Parameter, die vom Benutzer gesetzt werden könnten. Da ein Rückgabewert von *null* aber zum

Absturz oder zu Fehlfunktionen des GUI führen kann, wird einfach eine leere Parameterliste erzeugt und zurückgegeben. Für ein umfangreiches Beispiel einer Parameterliste vgl. Kap. B.1.4

public ListOfErrors configure(ParameterList aList)
Diese Methode wird nur für die Elemente der Pipeline aufgerufen, die vom Benutzer tatsächlich ausgewählt wurden. Der Parameter enthält in der Regel eine verkürzte Version der Parameterliste, die von *getParameterlist()* abgefragt wurde: Es befinden sich nur die Werte der Parameter in der Liste.

Der Aufbau dieser Methode orientiert sich häufig stark an der *getParameterlist()*-Methode. Das ist auch hier der Fall:

```
    public ListOfErrors configure(ParameterList aList) {
        return NoErrors.create();
    }
```

Codebeispiel 12: Die einfachste Möglichkeit, Parameterlisten auszuwerten

Da bei der **getParameterList()**-Methode keine Parameter zur Verfügung gestellt werden, werden auch keine zurückgeliefert.

Der Rückgabewert ist eine Liste von Fehlern, die beim Konfigurieren des Filters aufgetreten sind. Da der Filter nicht konfiguriert wird, können auch keine Fehler auftreten, aber auch hier führt ein die Rückgabe von **null** u.U. zum Absturz oder zu Fehlverhalten des Programms. Deswegen wird ein ”leerer” Fehler erzeugt und zurückgegeben. Für ein umfangreiches Beispiel vgl. auch hier Kap. B.1.4

public boolean filter()
Diese Methode wird aufgerufen, sobald **VisionControl** vom CONTROL-Modul das Signal bekommt, mit der Visualisierung zu beginnen. Hier soll die eigentliche Arbeit auf der **VisionDomain** stattfinden, in der Regel eine ”billige” Datenreduktion, um die nachfolgende ”teure” Visualisierung zu beschleunigen.

Zu dem Zeitpunkt an dem eine die **filter()**-Methode einer bestimmten Filter-Klasse aufgerufen wird, haben alle evtl. vorgeschalteten Filter ihre Arbeit an der **VisionDomain** bereits beendet, und die weitere Verarbeitung wird aufgeschoben, bis diese Methode erfolgreich beendet werden konnte. Die Klasse hat also exklusiven Zugriff auf die interne Datenstruktur.

Im Beispiel sieht diese Methode so aus:

```
    public boolean filter() {
        myLogger.debug("Identity filter doing nothing...");
        //Wow, this was fast :)
        return true;
    }
```

Codebeispiel 13: Die filter()-Methode des Identity-Filters

Der Rückgabe-Wert zeigt an, ob das Filtern erfolgreich war.

Das Attribut **myLogger** ist eine Instanz eines Log4j-Loggers (vgl. auch Kap.2.6.1), der bereits fertig konfiguriert ist.

Die Daten sind über die **VisionDomain**-Instanz **myDomain** zugänglich, die zu diesem Zeitpunkt ebenfalls fertig konfiguriert ist. Da der **Identity**-Filter die Daten nicht verändern muß, wird dieses Attribut hier gar nicht benutzt.

Abschließende Arbeiten
Damit das so auch funktioniert, müssen auch einige Klassen importiert werden:

```
import devisor.net.interim.ListOfErrors;
import devisor.net.interim.ParameterList;
import devisor.vision.util.AbstractFilter;
import devisor.vision.util.EmptyParameterList;
import devisor.vision.util.NoErrors;
```

Codebeispiel 14: Die Importe der neuen Filterklasse

Configurables.xml

Die zentrale Datei, in der alle Komponenten der Pipeline angemeldet werden müssen, ist die XML-Datei **Configurables.xml** im Paket **devisor.vision.util**. Sie wird als Basis benutzt, um alle verfügbaren Pipeline-Komponenten zu sammeln und ihre Parameterlisten zu erfragen.
Dies ist natürlich nicht die Stelle, um einen XML-Kurs durchzuführen (dafür wird auf aktuelle Literatur oder die Internet-Seite www.zvon.org verwiesen), aber das ist auch gar nicht notwendig, da die Datei extrem einfach strukturiert ist:
Für jede der drei Pipeline-Komponenten gibt es eine Auflistung, wie man hier sehen kann:

```
<configurables>

  <filterlist>
    <filter>
      <classname>
        devisor.vision.pipe.filter.IdentityFilter
      </classname>
    </filter>

    <!-- Hier können noch andere Einträge stehen -->
  </filterlist>


  <mapperlist>
    <mapper>
      <classname>
        devisor.vision.pipe.mapper.Arrow3DMapper
      </classname>
    </mapper>
```

```
    <mapper>
      <classname>
        devisor.vision.pipe.mapper.NullMapper
      </classname>
    </mapper>

    <!-- Hier können noch andere Einträge stehen -->
  </mapperlist>


  <rendererlist>
    <renderer>
      <classname>
        devisor.vision.pipe.renderer.SimpleRenderer
      </classname>
    </renderer>
  </rendererlist>


  <extralist>
    <extra>
      <classname>
        devisor.vision.movie.SnapshotTaker
      </classname>
    </extra>
    <extra>
      <classname>
        devisor.vision.movie.MovieMaker
     </classname>
    </extra>
  </extralist>

</configurables>
```

Es fällt auf, daß in diesem Kapitel kein Abschnitt für ”Extras” existiert. Das liegt daran, daß diese Extras nicht zur eigentlichen Visualisierungs-Pipeline gehören, sondern benutzt werden, wenn die Pipeline ihre Arbeit abgeschlossen hat. Es handelt sich dabei um den SnapshotTaker, der für die JPEG-Generierung der Ergebnisse zuständig ist, und um den MovieMaker, der aus den gesammelten Ergebnis-Bildern einen Film generiert.

Der Extras-Abschnitt ist nicht zur Erweiterung vorgesehen und sollte nicht verändert werden. Der Grund warum er sich trotzdem in dieser Datei befindet liegt darin, daß auch diese Extras konfiguriert werden können und Parameterlisten zur Verfügung stellen könnten.

Um unseren neuen Filter hinzuzufügen brauchen man nichts zu tun, weil es schon einen Eintrag für einen Filter mit dem Klassen-Namen **devisor.vision.pipe.filter.IdentityFilter** gibt. Ansonsten müssten nur drei Zeilen eingefügt werden, beginnend mit **<filter>** und endend mit **</filter>**. Dazwischen käme dann der Eintrag **<classname>voll.qualifizierter.Klassen-**

name</classname>.
Bei all dem sind die Einrückung wie auch der Zeilenumbruch irrelevant, es dient aber der Lesbarkeit, wenn der Stil der Datei eingehalten wird.

Die Internationalisierung

Um in den GUI-Elementen des CONTROL-Moduls den Namen und einige Beschreibungen unseres neuen Filters anzeigen zu können, muß zumindest noch die Datei **FindConfigurablesi18n.properties** im Paket **devisor.vision.util** bearbeitet werden. Hier befinden sich englischsprachigen Beschreibungen der Pakete. Die Einträge für die Pipeline-Komponenten befinden sich am Ende der Datei, und obwohl es egal ist, an welcher Stelle die neuen Einträge erfolgen kann es der Wartbarkeit des Programmpaketes zugute kommen, wenn die Einträge an den offensichtlichen Stellen erfolgen, d.h. dort wo auch die Einträge für die anderen Filter sind.
Die Einträge für den Filter sind auch hier wieder vorhanden, und an ihnen kann gut erkannt werden, welche Zeilen für Pipeline-Komponenten im Allgemeinen notwendig sind:

```
IdentityFilter.name=Identity Filter
IdentityFilter.shortname=IF
IdentityFilter.description=This filters nothing away.
```

Der Teil vor dem Gleichheitszeichen ist unschwer als Name der Klasse, die die Pipeline-Komponente enthält, zu erkennen, gefolgt von einem Punkt und einem der drei Schlüsselwörter **name**, **shortname** oder **description**. Für jede Pipeline-Komponente muß hier jedes der drei Schlüsselwörter genau einmal vorkommen. Der Teil nach dem Gleichheitszeichen definiert den Inhalt des entsprechenden Schlüsselwortes für diese Komponente.
Das Schlüsselwort **name** bekommt den englischen Namen dieser Klasse, **shortname** bekommt eine Abkürzung dieses Namens und **description** enthält eine Beschreibung der Pipeline-Komponente.
ACHTUNG: Die Einträge dürfen nur eine einzige Zeile einnehmen, die aber beliebig lang sein darf. Wenn Zeilenumbrüche z.B. in der Beschreibung gewünscht werden, so können sie durch die Zeichenfolge **\n** umschrieben werden.
Beschreibungen in weiteren Sprachen können in optionalen Dateien gemacht werden, die nach dem folgenden Muster benannt sein müssen: **FindConfigurablesi18n_de.properties**, wobei **de** das Sprachkürzel für Deutsch ist, und durch entsprechende andere Sprachkürzel ersetzt werden kann.
In diesen zusätzlichen Dateien müssen nicht alle drei Schlüsselwörter vorkommen. Falls eines fehlen sollte, wird auf den Eintrag in **FindConfigurablesi18n.properties** zurückgegriffen.
In *http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt* ist eine Liste der verfügbaren, gültigen Sprachkürzel zu finden.
Nun muß die entsprechende Klasse nur noch kompiliert werden und über den Klasspath erreichbar sein, dann wird beim nächsten Konfigurieren eines VISION-Moduls in der GUI auch die neue Komponente angezeigt.

B.1.2 Mapper

Um einen Mapper in die Pipeline zu integrieren, sind vier ähnliche Arbeitsschritte notwendig:

1. Das Schreiben einer Klasse, die das Interface **devisor.vision.pipe.mapper.I_Mapper** und das Interface **devisor.vision.I_Configurable** implementiert. Das passiert typischerweise, indem sie von der Klasse **AbstractMapper** aus demselben Paket erbt.

2. Die Implementierung der fehlenden Methoden.

3. Das Zufügen des neuen Pipeline-Elements zu der XML-Datei **devisor.vision.util.Configurables.xml**

4. Und zuletzt das Einfügen einiger Internationalisierungs-Strings zu der Standard-Datei **devisor.vision.util.FindConfigurablesi18n.properties**, sowie u.U. auch zu den anderen Internationalisierungs-Dateien in diesem Paket.

Der Mapper, der gebaut wird, wird nicht vollständig sein, weil auch der kürzeste Mapper in **devisor.vision.pipe.mapper** weit über 300 Zeilen lang ist. Der Beispiel-Code sollte aber ausreichen, um sich einen Überblick über die notwendigen Arbeiten zu verschaffen.

Die neue Mapper-Klasse

Eine neue Java-Klasse zu erzeugen, die von der Klasse **AbstractMapper** erbt, sieht zunächst einmal so aus:

```
public class uncompilableMapper extends AbstractMapper {

    public uncompilableMapper(){
        super();
    }

    protected BranchGroup buildSceneGraph(){    }

    public ParameterList getParameterList(){    }

    public ListOfErrors configure(ParameterList aList){    }
}
```

Codebeispiel 15: Grundgerüst jeder Mapperklasse

Die fehlenden Methoden

Diese Klasse ist noch nicht kompilierbar, weil noch einige Methoden zu implementieren sind:

public ParameterList getParameterList()
Für eine Erklärung dieser Methode vergleiche den Abschnitt darüber in Kap. B.1.1.
Siehe Kap. B.1.4 für ein umfangreiches Beispiel und Codebeispiel 11 für eine kurze Variante.

public ListOfErrors configure(ParameterList aList)
Für eine Erklärung dieser Methode vergleiche den Abschnitt darüber in Kap. B.1.1.
Siehe Kap. B.1.4 für ein umfangreiches Beispiel und Codebeispiel 12 für eine kurze Variante.

protected BranchGroup buildSceneGraph()
In dieser Methode geht die Übersetzung der GMV- und Gitter-Daten in einen SzeneGraphen (vgl. Kap. 2.1.2) vor sich.

Die Attribute **myLogger** und **myDomain** stehen auch hier zur Verfügung (vgl. B.1.1).

```
    protected BranchGroup buildSceneGraph() {
        BranchGroup bg = new BranchGroup();

        /* getExtremes() liefert einen Point2d mit
         * x als Minimum und y als Maximum */
        Point2d Extremes = getExtremes();

        // *********+ VALUE NODES +*********** //
        Enumeration nodeEnum = myDomain.getValueNodes()
            .elements();

        /*PointArray ist eine Klasse aus javax.media.j3d */
        PointArray myPointArray = new PointArray(
            myDomain.getValueNodes().size(),
            PointArray.COORDINATES | PointArray.COLOR_3
        );

        int i = 0;

        while (nodeEnum.hasMoreElements()) {
            ValueNode node = (ValueNode) nodeEnum.
                nextElement();

            Point3d myPoint = new Point3d(node.getCoord());
            Color3f myColor = getNodeColor(
                node.getScalarValue(), Extremes );

            myPointArray.setCoordinate(i, myPoint);
            myPointArray.setColor(i, myColor);

            i+=1;
        }

        Shape3D points = new Shape3D(myPointArray);
        points.setAppearance(getStandardAppearance());

        bg.addChild(points);

        /* Hier folgen noch diverse Zeilen, um neben
         * ValueNodes auch andere Elemente der
         * VisionDomain zu berücksichtigen */

        return bg;
    }
```

Codebeispiel 16: Ausschnitt aus der buildSceneGraph()-Methode einer Mapper-Klasse

Die Importe sind abhängig von der Implementierung des Mappers, aber auf jeden Fall werden die folgenden benötigt:

```
import devisor.net.interim.ListOfErrors;
import devisor.net.interim.ParameterList;
import devisor.vision.I\_Configurable;
import devisor.vision.ds.VisionDomain;
import devisor.vision.util.LegendLabel;
```

Codebeispiel 17: Die Importe der neuen Mapper-Klasse

Configurables.xml

Die Einträge in der Datei **Configurables.xml** im Paket **devisor.vision.util** wurden bereits in Kap. B.1.1 ausführlich beschrieben. Die dort für Filter beschriebenen Techniken können direkt auf Mapper übertragen werden.

Die Internationalisierung

Die Einträge in der Datei **FindConfigurablesi18n.properties** im Paket **devisor.vision.util** wurde bereits in B.1.1 ausführlich beschrieben. Die dort für Filter beschriebenen Techniken können direkt auf Mapper übertragen werden.

B.1.3 Renderer

Um einen Renderer in die Pipeline zu integrieren, sind vier ähnliche Arbeitsschritte notwendig:

1. Das Schreiben einer Klasse, die das Interface **devisor.vision.pipe.renderer.I_Renderer** und das Interface **devisor.vision.I_Configurable** implementiert. Das passiert typischerweise, indem sie von der Klasse **AbstractRenderer** aus demselben Paket erbt.

2. Die Implementierung der fehlenden Methoden.

3. Das Zufügen des neuen Pipeline-Elements zu der XML-Datei **devisor.vision.util.Configurables.xml**

4. Und zuletzt das Einfügen einiger Internationalisierungs-Strings zu der Standard-Datei **devisor.vision.util.FindConfigurablesi18n.properties**, sowie u.U. auch zu den anderen Internationalisierungs-Dateien in diesem Paket.

Der Renderer, der gebaut wird, wird die Funktion des **SimpleRenders** nachvollziehen, dem einzigen Renderer, der bisher im VISION-Modul existiert. Er sorgt für eine einfache Beleuchtung, setzt Grenzen für Sichtbarkeitsberechnungen und fügt die Reaktionen auf Mausbewegungen hinzu.

Die neue Renderer-Klasse

Für neue Java-Klasse, die von der Klasse **AbstractRenderer** erbt, vergleiche Codebeispiel 18.

Die fehlenden Methoden

Diese Klasse ist noch nicht kompilierbar, weil noch einige Methoden zu implementieren sind:

public ParameterList getParameterList()
Für eine Erklärung dieser Methode vergleiche den Abschnitt darüber in Kap. B.1.1 Siehe B.1.4 für ein umfangreiches Beispiel und Codebeispiel Kap. 11 für eine kurze Variante.

public ListOfErrors configure(ParameterList aList)
Für eine Erklärung dieser Methode vergleiche den Abschnitt darüber in Kap. B.1.1 Siehe Kap. B.1.4 für ein umfangreiches Beispiel und Codebeispiel Kap. 12 für eine kurze Variante.

public void setBranchGroup(BranchGroup aBranchGroup)
Diese Methode wird aufgerufen, wenn die Pipeline soweit durchlaufen ist, daß ein vollständiger Szenegraph (vgl. Kap. 2.1.2) berechnet wurde, damit nach ihrem Ende das Attribut **protected Canvas3D myCanvas3D** als Ergebnis vorliegt.

Die Attribute **myLogger** und **myDomain** stehen auch hier zur Verfügung (vgl. Kap. B.1.1).

```
/** Sets the scene to be rendered and (re-)initializes
 * the Canvas3D object on which rendering will be done
 * @param aBranchGroup
 */
public void setBranchGroup(BranchGroup aBranchGroup) {
    super.setBranchGroup(aBranchGroup);

    //This is inherited from AbstractRenderer
    configureCanvas();

    //Here we set the basics
    mySimpleUniverse = new SimpleUniverse(myCanvas3D);
    mySimpleUniverse.getViewer().getView().
        setBackClipDistance(10000d);
    mySimpleUniverse.getViewer().getView().
        setFrontClipDistance(.01d);
    mySimpleUniverse.getViewingPlatform().
        setNominalViewingTransform();


    myMouseConfig = myMouseManipulator.getMouseConfig();

    //creating transformations based on mouse movements
    myManipulator=new MouseManipulator(myBranchGroup);
    myBranchGroup =  myManipulator.
        getAttachedBranchGroup(myMouseConfig);
    myMouseConfig = myManipulator.getMouseConfig();
    myLogger.debug("MouseManipulator attached.");

    /*Adding transformation and own BranchGroup
     * to new Branchgroup-Object */
    BranchGroup rootBranchGroup = new BranchGroup();
    TransformGroup sceneTransform =
        new TransformGroup(myMouseConfig.getTransform());
    sceneTransform.addChild(myBranchGroup);
    rootBranchGroup.addChild(sceneTransform);
    myBranchGroup = rootBranchGroup;

    //Create a light and add it.
    addLight();

    myBranchGroup.compile();
    myLogger.debug("BranchGroup compiled.");

    mySimpleUniverse.addBranchGraph(myBranchGroup);
}
```

Codebeispiel 18: Die setBranchGroup-Methode einer Renderer-Klasse

Configurables.xml

Die Einträge in der Datei **Configurables.xml** im Paket **devisor.vision.util** wurden bereits in Kap. B.1.1 ausführlich beschrieben. Die dort für Filter beschriebenen Techniken können direkt auf Mapper übertragen werden.

Die Internationalisierung

Die Einträge in der Datei **FindConfigurablesi18n.properties** im Paket **devisor.vision.util** wurde bereits in Kap. B.1.1 ausführlich beschrieben. Die dort für Filter beschriebenen Techniken können direkt auf Mapper übertragen werden.

Abschließende Arbeiten

Die angeführten Methoden gehören in das Gerüst, das bereits im ersten Schritt erstellt wurde:

```
public class SimpleRenderer extends AbstractRenderer {

    public SimpleRenderer(){
        super();
    }

    public ParameterList getParameterList(){
    }

    public ListOfErrors configure(ParameterList aList){
    }

    public void setBranchGroup(BranchGroup aBranchGroup) {
    }
}
```

Codebeispiel 19: Grundgerüst jeder Rendererklasse

Die Importe sind natürlich abhängig von der Implementierung des Renderers, aber für unser Beispiel werden die folgenden benötigt:

```
import javax.media.j3d.BranchGroup;
import javax.media.j3d.TransformGroup;

import com.sun.j3d.utils.universe.SimpleUniverse;

import devisor.net.interim.ListOfErrors;
import devisor.net.interim.ParameterList;
import devisor.vision.util.EmptyParameterList;
import devisor.vision.util.MouseConfig;
import devisor.vision.util.MouseManipulator;
import devisor.vision.util.NoErrors;
```

Codebeispiel 20: Die Importe der neuen Mapper-Klasse

B.1.4 Eine Parameterliste

Parameterlisten bestehen aus ineinander verschachtelten Objekten, so daß Options-Hierarchien aufgebaut und Abhängigkeiten definiert werden können. Allerdings sind die Konstruktoren dieser Objekte recht unübersichtlich, was die Generierung von Parameterlisten zu einer schwierigen Aufgabe machen kann.

```
public ParameterList getParameterList() {
    //Internationalisierung wird konsequent praktiziert:
    ResourceBundle i18n =
        ResourceBundle.getBundle(
            "devisor.net.interim.interimi18n",
            LocaleHandler.getLocale());

    //Diese Liste wird später zurückgegeben
    ParameterList myList = new ParameterList(i18n);
    //Liste von Farben für Farbwähler
    EntryList myEntryList = new EntryList(i18n);
    myEntryList.addEntry(new Entry("Red", "255", i18n));
    myEntryList.addEntry(new Entry("Green", "255", i18n));
    myEntryList.addEntry(new Entry("Blue", "255", i18n));
    //Farbwähler für Quads
    VectorParameter quadParameter = new VectorParameter(
        4001, "Farbe von Quads",
        "QF", false,
        "Wählen Sie bitte die Farbe, in der "+
            "Quad-Objekte eingefärbt werden sollen",
        ScalarParameter.FORMAT_INTEGER,
        myEntryList, new ColorChooserDescription(i18n),
        i18n);
    //Zur Rückgabeliste
    myList.addParameter(quadParameter);
    //Hier folgen noch einige ähnliche Farbwähler
    //Ein-Ausschalter für Begrenzungen
    ScalarParameter showBoundaries = new ScalarParameter(
        4005, "Begrenzungen anzeigen?",
        "BA", false,
        "Bitte legen Sie fest, ob Begrenzungen "+
            "angezeigt werden sollen",
        ScalarParameter.FORMAT_BOOLEAN,
        "true", "N/A",
        "N/A", "N/A",
        new CheckBoxDescription(i18n),
        i18n);
    //Zur Rückgabeliste
    myList.addParameter(showBoundaries);

    return myList;
}
```

Codebeispiel 21: Die getParameterList()-Methode eines Mappers

```
public ListOfErrors configure(ParameterList aList) {
    ResourceBundle i18n =
        ResourceBundle.getBundle(
            "devisor.net.interim.interimi18n",
            LocaleHandler.getLocale());
    Enumeration paramEnum = aList.getEnumeration();
    ListOfErrors errors = new ListOfErrors(i18n);
    int errorCount = 0;
    while (paramEnum.hasMoreElements()) {
        Parameter p = (Parameter) paramEnum.nextElement();
        try {
            //show boundary as a grid ?
            if (p.getID()== 4005) {
                boolean choice = false;
                if (((ScalarParameter) p).getValue().
                    equalsIgnoreCase("true")) {
                    choice = true;
                }
            this.showBoundaryGrid(choice);
            //QuadColor
            } else if (p.getID()== 4001) {
                VectorParameter vParam=(VectorParameter) p;
                String redValue =
                    ((Entry) vParam.getList().getEntryAt(0)).
                        getValue();
                int red = Integer.parseInt(redValue);
                String greenValue =
                    ((Entry) vParam.getList().getEntryAt(1)).
                        getValue();
                int green = Integer.parseInt(redValue);
                String blueValue =
                    ((Entry) vParam.getList().getEntryAt(2)).
                        getValue();
                int blue = Integer.parseInt(redValue);
                Color aColor = new Color(red, green, blue);
                this.setQuadColor(aColor);
            }
        } catch (Exception e) {
            Error anError = new Error(i18n);
            anError.setID(errorCount);
            anError.setMessage(
            "Exception while configuring a DomainMapper:"
                + e.getMessage());
            errors.addError(anError);
            errorCount++;
        }
    }
    return errors;
}
```

Codebeispiel 22: Die configure-Methode eines Mappers

Diese **configure**-Methode (Codebeispiel 22) wird für die Elemente der Pipeline aufgerufen, die vom Benutzer tatsächlich ausgewählt wurden. Der Parameter enthält in der Regel eine verkürzte Version der Parameterliste, die von **getParameterList()** abgefragt wurde: Es befinden sich die Werte der Parameter in der Liste. Der Aufbau dieser Methode orientiert sich häufig stark an der **getParameterList()**-Methode.

B.2 Tutorial zur Integration eines neuen Moduls in das System

In diesem Abschnitt wird in Form eines Tutorials erläutert, wie ein neues Modul an den DEVISOR angebunden wird. Da insbesondere das Modul NET (s. Kap. 3.2) bereits Parser für das komplette Protokoll (s. Kap. D.3) zur Verfügung stellt, geschieht dies beispielhaft am **ExampleModule**, das als Referenz auch Teil der Distribution ist. Es befindet sich im Paket **devisor.net.example**.
Die Integration läßt sich folgendermaßen gliedern:

1. Definition der unterstützten Probleme und dazu notwendiger Parameterlisten,
2. Definition der unterstützten Protokollbefehle,
3. Implementierung des Wrappers für das neue Modul,
4. Implementierung des neuen Moduls und
5. Anbindung an den Server NEXUS

Diese Schritte werden im folgenden erklärt.

B.2.1 Definition der unterstützten Probleme und Parameterlisten

Das Beispielmodul wird sich nicht durch allzu viel Funktionalität auszeichnen, um nicht vom Sinn dieses Tutorials abzulenken: Es bietet lediglich die Möglichkeit, ausgehend von einem festen Startwert Integeradditionen in festen Zeitintervallen um einen festen Wert vorzunehmen und wahlweise Zwischenergebnisse sowohl als „Huckepack”-Statistik als auch per Ergebnisdownload zu liefern. Übersetzt in die Sprache der Interimsobjekte (s. Kap. 3.2) bedeutet dies, daß das Modul ein Problem definiert, und für dieses Problem drei skalare Parameter **initialValue**, **incrementValue**, **delayValue** existieren. Die globale Konfigurationsliste enthält einen Parameter, der als Flag den Umgang mit anfallenden Ergebnissen steuert sowie einen speziellen Parameter zur Konfiguration der Huckepack-Statistiken. Es sei daran erinnert, daß die Steuerung der Statistiken über die spezielle reservierte Parameter-ID 1 abgewickelt wird, und zwar mit einem **SelectionParameter**, der für jede zu erzeugende Statistik einen **ScalarParameter** enthält.

```
public class ExampleMC {
  public static ModuleCapabilities
    createMC(ResourceBundle i18n) {
    // create new container
    ModuleCapabilities mc=new ModuleCapabilities (i18n);
    // create problem ...
    // create parameter list
    ParameterList pl = new ParameterList (i18n);
    // create initial value parameter ...
    // create increment parameter ...
    // create delay parameter ...
    // and add them all to the parameter list
    pl.addParameter (initialvalue);
    pl.addParameter (incrementvalue);
    pl.addParameter (delayvalue);
    // register problem and its parameter list
    // with module capabilities
    mc.add (p,pl);
    // and create global conf list with one parameter
    ParameterList conf = new ParameterList (i18n);
    // create generateresult parameter ...
    conf.addParameter (generateresults);
    // create generateStatistics parameter ...
    conf.addParameter (generateStatistics);
    mc.setConfigurationList (conf);
    return mc;
  }
}
```

Codebeispiel 23: Rahmenklasse für die Problem- und Parameterspezifikation

Zur besseren Übersicht werden diese Deklarationen in einer eigenen Klasse vorgenommen, die lediglich eine statische Methode zur Verfügung stellt. Der Coderahmen kann Codebeispiel 23 entnommen werden. Es sei daran erinnert, daß alle Interimsklassen in ihren Konstruktoren eine Referenz auf ihre Lokalisierung übergeben bekommen, in den folgenden Codebeispielen ist sie immer mit **i18n** bezeichnet.
Im nächsten Schritt wird das Problem konfiguriert, wie Codebeispiel 24 zu entnehmen ist.

```
// create problem
Problem p = new Problem (
  // ID:
  PROBLEM_INCREMENTSIMULATOR,
  // name:
  "IncrementSimulator version V+",
  // description:
  A long-expected innovation in low performance\n"+
  "computing: just watch your integers skyrocket.\n"+
  "New in V+: now even with stock supervisor.",
  // tape recorder support and locale
  Problem.LOCAL_STARTSTOP, i18n);
```

Codebeispiel 24: Definition des Problems

Die Definition des Problems ist mit der Angabe der zugehörigen Parameterliste abgeschlossen (s. Codebeispiel 25). Es sei daran erinnert, daß alle Parameter-IDs unterhalb von 1024 reserviert sind. Insbesondere hat also der Parameter **GENERATESTATISTICS** die ID 1, weil er steuert, ob Statistiken Huckepack an das CONTROL-Modul transferiert werden sollen oder nicht.

```
// create initial value parameter
ScalarParameter initialvalue = new ScalarParameter (
  INCREMENTSIMULATOR_INITIALVALUE,    // ID
  "Initial Value",                    // name
  "IV",                               // short name
  false,                              // not configurable
  "integer in [-2^31..2^31-1",        // description
  ScalarParameter.FORMAT_INTEGER,     // format
  "0",                                // default value
  "$$$ in my stock portfolio",        // unit
  Integer.toString(Integer.MAX_VALUE),// max value
  Integer.toString(Integer.MIN_VALUE),// min value
  new TextFieldDescription(i18n),     // GUI
  i18n);                              // resource bundle
```

Codebeispiel 25: Definition eines Parameters

Codebeispiel 26 zeigt, wie ein Parameter zur Steuerung der Statistiken generiert wird.

```
ParameterList selectionlist = new ParameterList (i18n);
ScalarParameter stat = new ScalarParameter (5000,
  "current value",
  "VAL",
  false,
  "current value in the increment simulator",
  ScalarParameter.FORMAT_INTEGER,
  Integer.toString(0),
  "$",
  Integer.toString(Integer.MAX_VALUE),
  Integer.toString(Integer.MIN_VALUE),
  new TextFieldDescription (i18n),
  i18n);
selectionlist.addParameter (stat);
SelectionParameter generatestatistics =
  new SelectionParameter (
    INCREMENTSIMULATOR_GENERATESTATISTICS,
    "Generate statistics",
    "STAT",
    true,
    "Select Statistics you want to be generated",
    0,
    1,
    new SelectionDescription (i18n),
    selectionlist,
    selectionlist,
   i18n);
conf.addParameter (generatestatistics);
```

Codebeispiel 26: Parameter zur Statistiksteuerung

B.2.2 Definition der unterstützten Protokollbefehle

Die Definition der unterstützten Protokollbefehle ist noch einfacher durchzuführen: Die abstrakte Klasse **devisor.net.wrappers.AbstractCommunicationThread** muß einfach erweitert werden, und entsprechend einige Befehle blockiert werden. Codebeispiel 27 zeigt, wie dies umgesetzt werden kann.

```
public class ExampleCommunicationThread
      extends AbstractCommunicationThread {
  ...
  public void handleIncomingCommand (String command,
                BufferedReader din, PrintWriter dout)
                throws NetException {
    // first redefine some reactions
    try {
      // this module does not support these commands ...
      if (command.equals ("Step") ||
          command.equals ("FastForward") ||
          command.equals ("Rewind") ||
          command.equals ("Pause") ||
          command.equals ("GetCurrentStatistics") ||
          command.equals ("GetCompleteStatistics")) {
            // ... so we just reroute the command to the
            // unsupported"-handler which luckily is
            // defined in the superclass. In some other
            // cases, it might be useful to reroute to a
            // new method defined in this class.
            handleCommand_UnsupportedCommand (din, dout);
            return; // this call is crucial! It prevents the
                    // abstract comm thread from executing
                    // its original version of the
                    // command handlers.
      }
    } catch (Exception ex) {
      // if an exception comes this far up, send it
      // back via piggyback-ping
      if (!(ex instanceof NetException)) {
        // promote ex to NetException
        // to get the ListOfErrors
        ex = new NetException (ex,
                "ExampleCommunicationThread",
                "Exception in ExampleCommThread",
                "There were errors.",
                devisor.net.interim.Error.ERROR_COMMTHREAD,
                wrapper.getInterimBundle());
        // then add it to the queue
       wrapper.getOutbox().enqueue (
             ((NetException)ex).getListOfErrors(),
             wrapper.getProjectID(),
             Integer.toString(wrapper.getModuleID()),
             wrapper.getInstanceHandle());
    }
    // let the abstract class reply to all other commands
    super.handleIncomingCommand (command, din, dout);
  }
}
```

Codebeispiel 27: Blockierung von Protokollbefehlen

B.2.3 Implementierung des Wrappers für das neue Modul

Dieser Teil besteht im wesentlichen aus leichten Detailänderungen und Erweiterungen von existierenden abstrakten Klassen. In einer Endlosschleife werden ankommende Verbindungen akzeptiert und basierend auf dieser Verbindung eine neue Instanz des **ExampleCommunicationThreads** gestartet.

```
public class ExampleWrapper extends AbstractModuleWrapper {
  public String getProtocolSubtype () {
    // here we define the type of the module
    return Header.CONTROL_NUMERICS;
  }

  public void startCommunication () {
    // implementing this method is essential to
    // instanciate the correct threads
    try {
      while (true) {
        // block for incoming connections
        Socket socket = serversocket.accept();
        socket.setSoTimeout (TIMEOUT);
        // and just start the thread we just
        // defined one page ago
        new ExampleCommunicationThread (socket,
            "DeViSoR 1.0",this).start();
        }
    } catch (Exception ex) {
       // some sort of exception handling ...
    }
  }
}
```

Codebeispiel 28: Implementierung des Wrappers

B.2.4 Implementierung des neuen Moduls

Die Implementierung des neuen Moduls gestaltet sich etwas komplizierter, da sowohl Modul- als auch Netzwerkfunktionalität implementiert werden müssen. Kurz zusammengefaßt muß das Modul das Interface **devisor.net.wrappers.Wrappable** implementieren.

Zunächst müssen die nötigen Variablen deklariert werden, essentiell sind dazu Referenzen auf die eben definierte Klasse **ExampleWrapper**. Der Konstruktor setzt dann im wesentlichen nur Referenzen korrekt (s. Codebeispiel 29).

```
public class ExampleModule implements Wrappable {
   // reference to the wrapper itself, set via constructor
   private ExampleWrapper wrapper;
   // module capabilities, set via constructor
   private ModuleCapabilities mc;
   // module status, updated by various methods
   private ModuleStatus ms;
   // initial value, set via setCompleteConfiguration()
   private int initialValue;
   // increment value, set via setCompleteConfiguration()
   private int incrementValue;
   // delay value, set via setCompleteConfiguration()
   private int delayValue;
   // flag for results set via setCompleteConfiguration()
   private boolean generateResults;
   // flag for statistics set via setCompleteConfiguration()
   private boolean generateStatistics;
   // ResultList created by setCommand()
   private ResultList resultList;
   // vector of timesteps buffering the result list,
   // filled by inner class
   private Vector availableTimeSteps;
   // reference to inner class itself
   private Incrementor incrementor;

   /**
    *  the constructor just performs basic initialisations
    */
   public ExampleModule (ExampleWrapper wrapper) {
       this.wrapper = wrapper;
       this.mc = ExampleMC.createMC (
               wrapper.getInterimBundle());
       this.ms = new ModuleStatus (
         ModuleStatus.STATUS_STARTED, //started,unconfigured
         ModuleStatus.TAPERECORDER_STOP,
         0, // no timestep yet
         false); // no results available

   }
```

Codebeispiel 29: Konstruktor des Moduls

Danach geht es an die Implementierung der eigentlichen Modulfunktionalität, aus Gründen der Einfachheit wurde dies hier in einer inneren Klasse vorgenommen, um die Beispiele nicht durch zu viele Getter und Setter zu verkomplizieren. Die innere Klasse erweitert **TimerThread**, eine von Java zur Verfügung gestellte Klasse zur automatischen Wiederholung immer gleicher Aufgaben. Codebeispiel 31 gibt die Implementierung der wichtigsten Methode dieser Klasse an:

```
public void run () {
  // first perform increment and update status
  timestep++;
  ms.setTimeStep (timestep);
  value += incrementValue;
  // then generate result and statistics if neccessary
  if (generateResults) {
  // write result to file using the filename convention
  File file = new File
    ("INCREMENTSIMULATOR.result."+timestep);
  try {
    FileWriter out = new FileWriter (file);
    String s = "INCREMENTSIMULATOR - Result
      for timestep "+timestep+"\n";
    out.write (s,0,s.length());
    s = "Current Value: "+value;
    out.write (s,0,s.length());
    out.close ();
    // update result list
    availableTimeSteps.addElement (
      new TimeStep (timestep, "time", "date",
      file.length(),
      wrapper.getInterimBundle()));
    ms.setResultsAvailable (true);
  }
  catch (Exception ex) {
    // enqueue error message if that didn't work
  }
  // generate statistics
  if (generateStatistics) {
  ParameterList pl = new ParameterList (
    wrapper.getInterimBundle());
    pl.addParameter (new ScalarParameter (1,
                     "Progress",
                     "PG",
                     false,
                     "Progress of INCREMENTOR",
                     ScalarParameter.FORMAT_INTEGER,
                     Integer.toString (value),
                     "$$",
                     Integer.toString (value),
                     Integer.toString (value),
                     new TextFieldDescription(
                       wrapper.getInterimBundle()),
                     wrapper.getInterimBundle()));
  Statistics stat = new  Statistics ("date",
    "time", timestep, pl,
    wrapper.getInterimBundle());
```

Codebeispiel 30: Innere Klasse Incrementor

```
    wrapper.getOutbox().enqueue (stat,
      wrapper.getProjectID(),
      Integer.toString(wrapper.getModuleID()),
      wrapper.getInstanceHandle());
  }
}
```

Codebeispiel 31: Innere Klasse Incrementor

Wichtig: Zusätzlich zur eigentlichen Funktionalität muss lediglich Verwaltungsarbeit betrieben werden bzw. mit dem Kontrollmodul über die **outbox** des Wrappers kommuniziert werden.
Nun kann das Interface **Wrappable** implementiert werden. Dies erfordert jedoch nur noch Fleißarbeit, im wesentlichen müssen nur Referenzen aktualisiert bzw. Statusmeldungen generiert werden. Methoden, die bereits vom Wrapper abgeblockt werden, müssen nur mit einer leeren Standardimplementierung gefüllt werden. Als Beispiel folgt die Extraktion der Parametersetzungen in Codebeispiel 32.

```
public ListOfErrors setCompleteConfiguration (
        CompleteConfiguration cc) {
  // ignore problem, we just support one...
  // extract parameter values
  Enumeration params =
     cc.getParameterList().getEnumeration ();
  while (params.hasMoreElements ()) {
    ScalarParameter p =
          (ScalarParameter)params.nextElement();
    switch (p.getID()) {
       case ExampleMC.INCREMENTSIMULATOR_INITIALVALUE: {
       this.initialValue =
            Integer.parseInt (p.getValue ());
       break; }
       case ExampleMC.INCREMENTSIMULATOR_INCREMENTVALUE: {
         this.incrementValue =
             Integer.parseInt (p.getValue ());
         break; }
       case ExampleMC.INCREMENTSIMULATOR_DELAYVALUE: {
         this.delayValue =
             Integer.parseInt (p.getValue ());
         break; }
     }
   }
   // extract conf list ...
   // check if the settings make sense ...
   // generate status report ...
   // ... and return it, updating status first
   ms.setStatus (ms.STATUS_CONFIGURED);
}
```

Codebeispiel 32: Implementierung des Interfaces Wrappable

Um diese Klasse ausführbar zu machen, muß die **main**-Methode implementiert werden, hier

ist auf die korrekte Reihenfolge zu achten, um Nullpointer-Fehler zu vermeiden. Codebeispiel 33 gibt eine Implementierung an.

```
public static void main (String[] args) {
  if (args.length < 4) {
       // some sort of error message
  } else {
    // parse command line
    String projectID = args[0];
    int moduleID = Integer.parseInt(args[1]);
    String handle = args[2];
    int port = Integer.parseInt (args[3]);
    // and off we go
    ExampleWrapper wrapper = new ExampleWrapper (projectID,
                 moduleID, handle, port, null);
    ExampleModule module = new ExampleModule (wrapper);
    wrapper.setParent (module);
    wrapper.updateLocalization (wrapper.getHeaderLocale());
    wrapper.startCommunication ();
  }
}
```

Codebeispiel 33: Die main-Methode des Beispielmoduls

B.2.5 Anbindung an den Server NEXUS

Die Anbindung an den Server erfordert leider zur Zeit noch ein wenig Handarbeit:
Im ersten Schritt muß im Serververzeichnis, Unterverzeichnis **modules**, ein neues Verzeichnis mit dem Namen des Moduls erzeugt werden, in diesem Fall **modules_example**. In diesem Verzeichnis werden die folgenden beiden Dateien erstellt:

```
# change to workspace directory
cd $DEVISOR_WORKSPACE
# create new project directory
mkdir $1
# copy start script and utility script
cp $DEVISOR_SERVER/modules/
          module_example/start_examplemodule $1
cp -p $DEVISOR_SERVER/scan_resultfiles $1
```

Codebeispiel 34: Init-Skript des Beispielmoduls: ***init_examplemodule***

```
# for parameters see class ExampleModule
java -cp $DEVISOR
      devisor.net.example.ExampleModule $1 $2 $3 $4
```

Codebeispiel 35: Start-Skript des Beispielmoduls: ***start_examplemodule***

Für die hier verwendeten Umgebungsvariablen wird auf den Abschnitt „Installation” (s. Kap. A.2) verwiesen. Wichtig: beide Skripte müssen ausführbar sein.

Nun muss das Modul noch im Server eingetragen werden. Dazu ist die Datei **server.c** aus dem Server-Verzeichnis zu editieren, und dem Feld der Module ein neuer Eintrag hinzuzufügen. Die einzelnen Einträge sind Codebeispiel 36 zu entnehmen.

```
// next free ID, check thoroghly:
modules[4].moduleid=4;
// name:
strcpy(modules[4].name,"Example");
// type:
modules[4].type=MODULETYPE_NUMERICS;
// description
strcpy(modules[4].descr,"Example module");
// start-script
strcpy(modules[4].appl,"./start_examplemodule ");
// init-script
strcpy(modules[4].initappl,
 "./modules/module_example/init_examplemodule ");
// server shall pipe commands through
modules[4].modus=TRUE;
```

Codebeispiel 36: Eintragen des Beispielmoduls in den Server

Nach einer kompletten Neukompilierung des Servers und des Moduls ist die Integration abgeschlossen.

B.3 Einbindung anderer Programmpakete

Um andere Programmpakete in das DEVISOR-Framework zu integrieren, sind Modifikationen am Server NEXUS notwendig. Dies soll im folgenden anhand des FEATPOISSON-Moduls exemplarisch dargestellt werden.Die Beispiele erfolgen in der Sprache C und nicht in Java.
Zuerst muß eine Kennung festgelegt und in `devisor/server/server/parser.h` definiert werden, z.B `FEATPOISSON`. Die Konstante `MAX_MODULE` muß entsprechend erhöht werden (Codebeispiel 37).

```
#define MODULE_TYPE_FEATPOISSON 4
#define MAX_MODULE 8
```

Codebeispiel 37: Modulkennung

Im Quelltextbaum muß ein Modulerzeichnis festlegt werden, z.B.

`/devisor/server/modules/module_fp`

Die Aktivierung der verschiedenen Module wird über eine Konfigurationsdatei gesteuert. Die Steuerung der Aktivierung befindet sich in der Datei
`devisor/server/server/server.c` in der Prozedur
`void init_parameterlist()`.
Codebeispiel 38 zeigt das Codefragment zur Aktivierung des Moduls.

```
else if (strcmp(rl,"FEATPOISSON\n")==0)
{
  printf("Module FEATPOISSON enabled.\n");
  enablemodule[MODULE_TYPE_FEATPOISSON]=1;
}
```

Codebeispiel 38: Modul-Aktivierung

In Codebeispiel 39 ist das Setzen der Moduldefinitionsvariable `modules` dargestellt. `modules` ist eine C-Struktur und enthält folgende Komponenten:

- `moduleid`: fortlaufende Nummer,
- `callsign`: die eingangs vergebene Identifikationsnummer,
- `name`: Kurzname des Moduls,
- `descr`: Beschreibung des Moduls,
- `appl`: Name der zu startenden Programmdatei,
- `initappl`: Name eines Initialisierungsskriptes. Dieses Skript erzeugt die Verzeichnisse des Projektes und kopiert Modul- und Hilfsprogramme dorthin. Als Beispiel können die Skripte der anderen Module dienen.
- `modus`: Definition des Modulhandling, für externe Module ist hier immer der Wert `PARSER_MODUS_NOPIPE_INDIRECT` zu setzen.

```
if (enablemodule[MODULE_TYPE_FEATPOISSON]==1)
{
  modules[mc].moduleid=mc;
  modules[mc].callsign=MODULE_TYPE_FEATPOISSON;
  strcpy(modules[mc].name,"FeatPoisson");
  modules[mc].type=MODULETYPE_NUMERICS;
  strcpy(modules[mc].descr,
    "FEAT2D-Testmodul Poisson mit LOCUTUS-Linkmodul");
  strcpy(modules[mc].appl,"./feat2dtest ");
  strcpy(modules[mc].initappl,
    "./modules/module_featpoisson/init_module3 ");
  modules[mc].modus=PARSER_MODUS_NOPIPE_INDIRECT;
```

Codebeispiel 39: Moduldefinition

Für ein Modul können mehrere Probleme definiert werden. Ein Problem setzt sich aus der Definition der Parameter, der Statistikwerte und der Resultate zusammen. Die Parameter werden in einer Parameterdatei definiert, die Übergabe an das Programm ist aber sehr programmspezifisch und muß deshalb im Server selbst kodiert werden. Fuer die Parameter können alle Typen benutzt werden, die das DEVISOR-Framework vorsieht.
In der `modules`-Struktur ist ein Array `prb` vorhanden, welches die möglichen Probleme definiert. Die Variable `maxprb` gibt die Anzahl der definierten Probleme an. Codebeispiel 40 zeigt die Definition des Membranproblems. Die Komponente `lp` gibt an, welche Steuerungsmöglichkeiten das Modul erlaubt.

```
modules[mc].maxprb=1;
modules[mc].prb[0].problemid=0;
strcpy(modules[mc].prb[0].name,"Membran");
strcpy(modules[mc].prb[0].descr,"Membran");
modules[mc].prb[0].lp=PARSER_LOCALPROTO_FULL;
```

Codebeispiel 40: Problemdefinition

Die Ergebnisse, die das Modul liefern kann, werden in einer verketteten Liste definiert, deren Elemente vom Typ `tresult` sind. In jedem Element werden Indexnummer, Typ, Beschreibung, lokaler Dateiname, d.h. unter welchem Namen die Ergenisse vom Modul ins lokale Arbeitsverzeichnis geschrieben werden gespeichert. Für Ergebnisse vom Typ GMV werden zusätzlich die in dieser Datei vorhandenen Datenfelder angegeben. Die Namen müssen exakt mit denen in der Datei vorkommenden übereinstimmen. Codebeispiel 41 zeigt ein exemplarisches Codefragment.

```
modules[mc].prb[0].result=&bspresult1;

bspresult1.resultid=0;
bspresult1.resulttype=RESULT_TYPE_GMV;

bspresult1.resultname=(char*)malloc(32);
strcpy(bspresult1.resultname,"Auslenkung");

bspresult1.resultdescr=(char*)malloc(32);
strcpy(bspresult1.resultdescr,"Auslenkung");

bspresult1.resultmask=(char*)malloc(32);
strcpy(bspresult1.resultmask,"result.auslenkung.gmv");

bspresult1.avail=0;
bspresult1.ndatafields=1;
strcpy(bspresult1.datafields[0],"solution");

bspresult1.result_timestep=NULL;
bspresult1.result_size=NULL;

bspresult1.next=NULL;
```

Codebeispiel 41: Ergebnisdefinition

Die Parameter, die ein Problem definieren, werden in einer separaten Datei gespeichert. Die Parameterdatei wird durch einen Aufruf der Funktion `load_parameter_file` eingelesen und damit der Problemdefinition hinzugefügt (s. Codebeispiel 42).

```
  modules[mc].prb[0].param=(tpara*)malloc(sizeof(tpara));
  load_parameter_file(modules[mc].prb[0].param,
    "./modules/module_featpoisson/test2.param");
```

Codebeispiel 42: Einlesen der Parameter

Codebeispiel 43 zeigt eine Parameterdatei für das FeatPoisson-Problem.

Das erste Schlüsselwort beschreibt den Typ des Parameters (ScalarParameter, GridParameter, FunctionParameter, SelectionParameter oder NodeParameter). Dann folgen Indexnummer, Parametername, Kurzname und Angabe der Rekonfigurierbarkeit. Für SelectionParameter folgen Minimalauswahl, Maximalauswahl und Anzahl der folgenden Subparameter, für Nodeparameter nur die Anzahl der folgenden Subparameter. Für alle Parametertypen folgt anschließend die Beschreibung, die durch die Zeichenfolge "`---`" beendet wird. Die weiteren Einträge hängen vom Typ ab. Bei Node- und Selectionparametern folgen die Subparameter, Scalarparameter werden durch Typ (Integer oder Double), minimaler erlaubter Wert, maximal erlaubter Wert, Defaultwert und Einheit definiert. GridParameter werden durch Eingabe des Gitters im FEAST-Format definiert. Diese Angabe muß wieder durch `---` beendet werden. Bei Function-Parameter folgen die globalen Parameter und die Funktionsdefinition. Abgeschlossen wird die Parameterdefinition durch die Angabe des zu verwendenden GUI-Elements.

```
SelectionParameter,1,Statistics,STAT,FALSE,1,3,1
This parameter defines the available statistic items
---
ScalarParameter,1038,Iteration,ITER,FALSE
Count of iterations
---
Integer,0,10000,0,nonunit
GUI,TextField

=======================================================
GUI,CheckBox

NodeParameter,1037,Parameters,PARAM,FALSE,6
This parameter defines the available parameters
---
ScalarParameter,1027,RefinementLevel,NFINE,FALSE
This parameter sets the number of refinements
---
Integer,1,10,3,nonunit
GUI,TextField

ScalarParameter,1032,RelativeAccuray,EPS CG,TRUE
If this relative accurac is fulfilled the iteration stops
---
Double,1E-6,1E-16,1E-6,nonunit
GUI,TextField
GUI,CheckBox
```

Codebeispiel 43: Parameterdatei Teil 1

```
GridParameter,1034,Gitter,GRID,FALSE,6000
Der Startwert
---
FEAST
....
#
---
GUI,GridViewer

FunctionParameter,1035,Righthandside,RHS,FALSE,6,1
Definition of the right hand side
---
Double: x
Double: y
Integer: ia
Integer: ida
Integer: bfirst
Double: dt
---
rhs:=-1.0-0.1*dt;
---
GUI,FunctionEditor

ScalarParameter,1041,Maximum Number of global Iterations,
MAXNITER,TRUE
Maximum Number of global Iterations
---
Integer,0,100,10,nonunit
GUI,TextField

ScalarParameter,1042,Maximum storage block,NNWORK,FALSE
Maximum storage block for the program in int
---
Integer,0,2000000000,50000000,nonunit
GUI,TextField

=======================================================
GUI,CheckBox
```

Codebeispiel 44: Parameterdatei Teil 2

Auf die Parameter sollte das Tool `create_pathfiles` angewendet werden, welches Headerdateien für C und Fortran mit den Konstanten für die Parameterids erzeugt.

In der Datei `devisor/server/server/server.c` muß an der Stelle `/** MODULE PARAMETER HANDLING **/` der Code für die initiale Konfiguration angegeben werden. Codebeispiel 45 zeigt das entsprechende Codesegment für das FeatPoisson-Problem.

```
else if (modules[command.moduleid].callsign
      ==MODULE_TYPE_FEATPOISSON)
{

  // generation of grid file "test"

  par=routertable[actrtl].apara;
  while (par!=NULL)
  {
    if ((par->parameterid>1024) && (par->parameterid<4096))
    {
      if (par->type==PARA_GRID)
      {
        strcpy(buffer,routertable[actrtl].wdpath);
        strcat(buffer,"test");

        transfer_grid(par,buffer);

      }
    }

    par=par->next;
  }
```

Codebeispiel 45: Parameterkonfiguration 1

```
  // generation of include file "nnwork.inc"

  strcpy(buffer,routertable[actrtl].wdpath);
  strcat(buffer,"nnwork.inc");
  fh=fopen(buffer,"w");
  fprintf(fh,
    "      PARAMETER (NNARR=299,NNAB=21,NNWORK=%ld)\n",
    parper[LVD_FEPO_NNWORK-PLMIN]);
  fclose(fh);
```

Codebeispiel 46: Parameterkonfiguration 2

```
// generation of parameter file "TEST2D.DAT"

strcpy(buffer,routertable[actrtl].wdpath);
strcat(buffer,"TEST2D.DAT");

fh=fopen(buffer,"w");
fprintf(fh,"%i\t\tM\n",parper[LVD_FEPO_M-PLMIN]);
fprintf(fh,"%i\t\tMT\n",parper[LVD_FEPO_MT-PLMIN]);
fprintf(fh,"%i\t\tNFINE\n",parper[LVD_FEPO_NFINE-PLMIN]);
fprintf(fh,"%i\t\tICUB\n",parper[LVD_FEPO_ICUB-PLMIN]);
fprintf(fh,"%i\t\tICUB RHS\n",
  parper[LVD_FEPO_ICUB_RHS-PLMIN]);
fprintf(fh,"%i\t\tISOLV CG\n",
  parper[LVD_FEPO_ISOLV_CG-PLMIN]);
fprintf(fh,"%i\t\tNIT CG\n",parper[LVD_FEPO_NIT_CG-PLMIN]);
fprintf(fh,"%E\t\tEPS CG\n",
  parper1[LVD_FEPO_EPS_CG-PLMIN]);
fprintf(fh,"%E\t\tOMEGA CG\n",
  parper1[LVD_FEPO_OMEGA-PLMIN]);
fprintf(fh,"2             ICUBM ELIT\n");
fprintf(fh,"8             ICUB2 ELPT\n");
fprintf(fh,"%i\t\tMAXNITER\n",
  parper[LVD_FEPO_MAXNITER-PLMIN]);
fprintf(fh,"66            OUTPUT UNIT\n");
fprintf(fh,"'TEST2D.OUT'   OUTPUT FILENAME\n");

fclose(fh);
```

Codebeispiel 47: Parameterkonfiguration 3

```
  // generation of source file "bdry.f"

  strcpy(buffer,routertable[actrtl].wdpath);
  strcat(buffer,"bdry.f");

  fh=fopen(buffer,"w");

  par=routertable[actrtl].apara;
  while (par!=NULL)
  {
    if ((par->type==PARA_FUNCTION) &&
      (par->parameterid==LVD_FEPO_RHS))
    {
      fprintf(fh,
"  DOUBLE PRECISION FUNCTION RHS(X,Y,IA,IDA,BFIRST,DT)\n");
      fprintf(fh,
"  IMPLICIT DOUBLE PRECISION(A,C-H,O-U,W-Z),LOGICAL(B)\n");
      convert_function(fh,par);
      fprintf(fh,"      END\n");
    }

    par=par->next;
  }

  fclose(fh);

}
```

Codebeispiel 48: Parameterkonfiguration 4

Als letzter Abschnitt werden die Statistikwerte definiert. Statistikwerte können skalare Integer- oder Double-Werte sein. Analog zu den Resultaten werden sie durch eine Struktur `tstat` definiert. Codebeispiel 49 zeigt eine solche.

```
bspstat.anzstat=3;

bspstat.stype[0]=STAT_TYPE_SCALARINT;
strcpy(bspstat.sname[0],"Number of Iterations");
strcpy(bspstat.sshortname[0],"NIT");

bspstat.stype[1]=STAT_TYPE_SCALARDOUBLE;
strcpy(bspstat.sname[1],"Convergance rate");
strcpy(bspstat.sshortname[1],"CONVRATE");

bspstat.stype[2]=STAT_TYPE_SCALARDOUBLE;
strcpy(bspstat.sname[2],"Calculation time");
strcpy(bspstat.sshortname[2],"TIME");

modules[mc].prb[0].stat=&bspstat;
```

Codebeispiel 49: Definition der Statistiken

Wichtig, die Reihenfolge und der Typ der Statistikwerte müssen exakt mit der Sequenz der `open_stat`, `put_stat` und `close_stat` Befehle im eigentlichen Modul übereinstimmen, da ansonsten die richtige Zuordnung nicht möglich ist und falsche Statistikwerte geliefert werden.
Damit sind die serverseitigen Anpassungen vollständig, anschließend folgt die Anpassung des Moduls. Generell läßt sich sagen, daß sich ein vorhandenes Programm ohne irgendwelche Änderungen einbinden läßt. Allerdings sind dann die Steuerungsmöglichkeiten, insbesondere die Kassettenrekorderfunktion, sehr begrenzt.
Das folgende Programm, implementiert in FORTRAN ist eine einfache Applikation, welche die Lösung des Poissonproblems mit Hilfe des FiniteElemente-Paktes FEAT2D berechnet.

```
      PROGRAM FEAT2DTESTQ
      IMPLICIT DOUBLE PRECISION(A,C-H,O-U,W-Z),LOGICAL(B)
      include "nnwork.inc"
      PARAMETER (NBLOCA=1,NBLOCF=1)
      CHARACTER SUB*6,FMT*15,CPARAM*120
      CHARACTER CFILE*60,ARRDA*6,ARRDF*6
      CHARACTER CFILE1*15
      DIMENSION VWORK(1),KWORK(1)
      DIMENSION KABA(2,NNAB,NBLOCA),KF(NNAB,NBLOCF)
      DIMENSION KABAN(NBLOCA),KFN(NBLOCF)
      DIMENSION BCONA(NBLOCA),BCONF(NBLOCF)
      DIMENSION LA(NBLOCA),LF(NBLOCF)
      DIMENSION ARRDA(NBLOCA),ARRDF(NBLOCF)
      DIMENSION BSNGLA(NBLOCA),BSNGLF(NBLOCF)
      COMMON          NWORK,IWORK,IWMAX,L(NNARR),
     *                DWORK(NNWORK)
      COMMON /ERRCTL/ IER,ICHECK
      COMMON /CHAR/   SUB,FMT(3),CPARAM
      COMMON /TRIAD/  NEL,NVT,NMT,NVE,NVEL,NBCT,NVBD
      COMMON /TRIAA/  LCORVG,LCORMG,LVERT,LMID,
     *                LADJ,LVEL,LMEL,LNPR,LMM,
     *                LVBD,LEBD,LBCT,LVBDP,LMBDP
      COMMON /OUTPUT/ M,MT,MKEYB,MTERM,MERR,MPROT,
     *                MSYS,MTRC,IRECL8
      EQUIVALENCE (DWORK(1),VWORK(1),KWORK(1))
      EXTERNAL PARX,PARY,TMAX
      EXTERNAL COEFFA,RHS,UE
      EXTERNAL S2DI0,S2DB0
      EXTERNAL E011,I000
      DATA KABAN/2/,KFN/1/
      DATA BCONA/.TRUE./,BCONF/.FALSE./
      DATA ARRDA/'DA    '/,ARRDF/'DF    '/
      DATA BSNGLA/.FALSE./,BSNGLF/.FALSE./
      INTEGER isstart,ispause
      INTEGER IRET,ISOCKET,IPAR,MAXNITER,IRSOCKET
      DOUBLE PRECISION DPAR
      LOGICAL FIRST
```

Codebeispiel 50: Beispiel: FeatPoisson Initialisierung

Codebeispiel 50 stellt den Initialisierungsblock dar.

```
      include "locutusf.h"

      include "fpardef.h"

C
C *** Initialization
C
      FIRST=.TRUE.

      call getarg(1,CFILE1)
      read(CFILE1,"(I5)") IH

      call getarg(2,CFILE1)
      read(CFILE1,"(I5)") ISOCKET

      call getarg(3,CFILE1)
      read(CFILE1,"(I5)") IRSOCKET

      call dllm_init_relink(IH,ISOCKET,IRSOCKET,IRET)

      call dllm_reset_status()
```

Codebeispiel 51: Beispiel: FeatPoisson Initialisierung

Die drei Parameter Socket, Remotesocket und Instancehandle werden uber die Kommandozeile eingelesen und die Funktion `dllm_init_relink` wird mit diesen Parameterm aufgerufen, um die vom VINCULUM-Modul initierte Verbindung benutzen zu können, um das Ergebnis der Initialisierung an den aufrufenden Serverprozeß melden zu können.
Im folgenden Codeabschnitt wird die Initialisierung des Testproblems mit Hilfe der FEAT2D-Routinen durchgeführt. Die Datei mit den Parameter wird eingelesen und die Routinen zur Initialisierung der Datenstrukturen werden aufgerufen. Im Fehlerfall wird eine Fehlermeldung zurückgeliefert und das Programm beendet.

```
      if (first.eqv..TRUE.) then
        first=.FALSE.
        ier=0
        call dllm_return_feat_errcode(ier)
      endif
```

Codebeispiel 52: Beispiel: FeatPoisson Rückmeldung

Wenn das Programm bei der Initialisierung bis an diese Stelle gekommen ist (Codebeispiel 52), ist alles ok und dies wird dem aufrufenden Server mitgeteilt.

```
1000  CONTINUE
      call dllm_listen_link(ITIME,ICMD,IVAL,IPAR,DPAR)
```

Codebeispiel 53: Beispiel: FeatPoisson Hauptschleife

Das ist die Hauptschleife. Die Routine fragt den lokalen Port ab und liefert ggf. das übertragene Kommando nebst den Integerparametern IVAL, IPAR und dem Doubleparameter DPAR zurück.

```
      if (ICMD.EQ.LOCUTUS_SETINTPAR) then
        print *,"Changing value ",IVAL," to ",IPAR

        if (IVAL.EQ.LVD_FEPO_MAXNITER) MAXNITER=IPAR
        if (IVAL.EQ.LVD_FEPO_NIT_CG) NIT=IPAR

      endif

      if (ICMD.EQ.LOCUTUS_SETDOUBLEPAR) then
        print *,"Changing value ",IVAL," to ",DPAR

        if (IVAL.EQ.LVD_FEPO_EPS_CG) EPS=DPAR
        if (IVAL.EQ.LVD_FEPO_OMEGA) OMEGA=DPAR

      endif
```

Codebeispiel 54: Beispiel: FeatPoisson Hauptschleife

Codebeispiel 54 realisert das dynamische Setzen der Parameter. In IVAL wird die ID des gewünschten Parameters übergeben und in IPAR rsp. DPAR der Wert des Parameters.

```
      if (ICMD.EQ.LOCUTUS_REWIND) then
        print *,"FEAT2DTEST rewind"
        itime=itime-ival
        DT=DT-ival*0.5D0
        if (itime.lt.0) itime=0
        if (dt.lt.0D0) dt=0D0
        call dllm_update_stat(itime)
      endif
```

Codebeispiel 55: Beispiel: FeatPoisson Hauptschleife

Codebeispiel 55 implementiert die Funktion „Zurückspulen“.

```
      IVAL=0
      call dllm_listen_return(ICMD,IVAL)
```

Codebeispiel 56: Beispiel: FeatPoisson Hauptschleife

An dieser Stelle (Codebeispiel 56) wird die Rückmeldung auf den übermittelten Befehl übermittelt. Sollte kein Kommando übermittelt worden sein, wird auch keine Antwort generiert. Dies macht die Routine automatisch.

```
      if (ICMD.EQ.LOCUTUS_STOP) then
          print *,"FEAT2DTEST stop"
          goto 1
      endif
```

Codebeispiel 57: Beispiel: FeatPoisson Hauptschleife

Codebeispiel 57 implementiert die Funktion „Stop“.

```
      call is_running(isstart)
      call is_paused(ispause)

      if ((ICMD.EQ.LOCUTUS_STEP).or.
     * ((isstart.eq.1).and.(ispause.eq.0))) THEN
```

Codebeispiel 58: Beispiel: FeatPoisson Hauptschleife

An dieser Stelle (Codebeispiel 58) wird der Status des Programms abgefragt. Nur wenn das Step-Kommando erfolgt ist, oder wenn das Programm gestartet wurde und sich nicht im Pausemodus befindet, wird ein Rechenschritt ausgeführt.

```
      CALL ZTIME(TIME)
      WRITE (MPROT,*) 'TIME PREPARATION ',TIME

      CALL XLL21(LF(1),NEQ,RBNORM)
      EPS=EPS*RBNORM

      IF (ISOLV.EQ.1)
     * CALL XIE017(LA(1),LCOLA,LLDA,LU,LF(1),
     *             NEQ,NIT,ITE,EPS,OMEGA,I000)
      IF (ISOLV.EQ.2)
     * CALL XII017(LA(1),LCOLA,LLDA,LU,LF(1),
     *             NEQ,NIT,ITE,EPS,OMEGA,I000)
      IF (ISOLV.EQ.3)
     * CALL XIR017(LA(1),LCOLA,LLDA,LU,LF(1),
     *             NEQ,NIT,ITE,EPS,OMEGA,I000)
      IF (ISOLV.EQ.4)
     * CALL XIS017(LA(1),LCOLA,LLDA,LU,LF(1),
     *             NEQ,NIT,ITE,EPS,OMEGA,I000)
C
C *** The cpu time for the solution is displayed
      CALL ZTIME(TIME)
      WRITE (MPROT,*) 'TIME SOLVER ',TIME

      if (itime.lt.10) then
        write(CFILE,"(A,I1)") "result.auslenkung.gmv.",ITIME
      else if (itime.lt.100) then
        write(CFILE,"(A,I2)") "result.auslenkung.gmv.",ITIME
      else if (itime.lt.1000) then
        write(CFILE,"(A,I3)") "result.auslenkung.gmv.",ITIME
      else if (itime.lt.10000) then
        write(CFILE,"(A,I4)") "result.auslenkung.gmv.",ITIME
      else if (itime.lt.100000) then
        write(CFILE,"(A,I5)") "result.auslenkung.gmv.",ITIME
      else if (itime.lt.1000000) then
        write(CFILE,"(A,I6)") "result.auslenkung.gmv.",ITIME
      else
        write(CFILE,"(A,I7)") "result.auslenkung.gmv.",ITIME
      endif

      MUNIT=57
      CALL  XGMV2D(MUNIT,CFILE,NEL,NVT,KWORK(L(LVERT)),
     *             DWORK(L(LCORVG)),DWORK(L(LU)))

      call result_available
```

Codebeispiel 59: Beispiel: FeatPoisson Hauptschleife

Codebeispiel 59 zeigt, wie die Lösungsroutine aufgerufen, und das Ergebnis als GMV-Datei herausgeschrieben wird. Außerdem wird durch Aufrufen der Routine `result_available` dem lokalen Server mitgeteilt, daß Resultate verfügbar sind. Dies muß manuell vom Modul selbst erledigt werden.

```
      call dllm_open_stat(ITIME)
      call dllm_put_stat_int(ITE)
      call dllm_put_stat_double(EPS)
      call dllm_put_stat_double(DTIME)
      call dllm_close_stat
```

Codebeispiel 60: Beispiel: FeatPoisson Statistikgenerierung

Die Statistikfile für diesen Zeitschritt wird geöffnet und die Statistikwerte werden in der in der Parameterdatei festgelegten Reihenfolge herausgeschrieben. Danach wird die Datei wieder geschlossen (Codebeispiel 60).

```
      if (ICMD.eq.LOCUTUS_EXIT) goto 3001
      if (ITIME.gt.MAXNITER) goto 1

3000  GOTO 1000
3001  CONTINUE
```

Codebeispiel 61: Beispiel: FeatPoisson Ende der Hauptschleife

Am Ende der Hauptschleife (s. Codebeispiel 61) werden Vorbereitungen für den nächsten Zeitschritt getroffen und die Endbedingen überprüft.

```
      call dllm_link_exit()

      GOTO 99999

99998 WRITE(MTERM,*) 'IER', IER
      WRITE(MTERM,*) 'IN SUBROUTINE ',SUB

      call dllm_return_feat_errcode(ier)

99999 END
```

Codebeispiel 62: Beispiel: FeatPoisson Programmende und Fehlerbehandlung

Codebeispiel 62 stellt den Code zur Programmbeendigung und zur Fehlerbehandlung dar.

B.4 Ein neues Element in GRID erstellen

Hier wird exemplarisch beschrieben, was man tun muß, um ein neues 3D-Zeichenelement (in unserem Beispiel ein Cone (Kegel)) in GRID hinzuzufügen.

B.4.1 Modifikation am FEAST-Format

Ein Kegel ist ein Randbeschreibungsobjekt und besteht wie ein Zylinder aus zwei Punkten (Nodes) und einem Radius. Damit dieser gespeichert wird, muß das FEAST-Format zunächst

geändert werden. Dies ist eine recht umfassende Änderung, die in mehreren Modulen durchgeführt werden muß. Hier wird beschrieben, wie man die Formatänderung in Grid und in den Interims-Objekten durchführt. Die andere Seite der Kommunikation, also vor allem der Server NEXUS und das Numerikmodul bleiben außen vor.
Um sich mit dem FEAST-Format vertraut zu machen, empfiehlt sich das Studium der entsprechenden Passagen dieses Endberichts (siehe Kap. D.1). An der Stelle, in der die Boundary3D-Objekte definiert werden, muß die folgende Sequenz zur Beschreibung einer Cone definiert werden (Codebeispiel 63):

```
4              # Typ 5  Kegel
0.0 0.0 0.0    # Fußpunkt
5.0            # Radius
5.0 5.0 5.0    # Normalenvektor=Fußpunkt-Zweiter Punkt
2.5 2.5 2.5    # Mittelpunkt
```

Codebeispiel 63: Erweiterungsvorschlag für ein neues 3D-Boundary-Objekt, Cone

Dieses neue Element muß auch noch in dem Parser-Interim-Objekt
devisor.net.interim.GridParameter eingetragen werden und zwar sowohl in die Methode **feastToDomain** als auch in **domainToFeast**. Das sind die Methoden, die Strings im FEAST-Format in die Domain-Datenstruktur umwandeln und umgekehrt. Dort muß bei der 3D-Behandlung, bei den Boundaries genauso wie für **BoundaryCylinder** verfahren werden.

B.4.2 Modifikation an der Domain

Da es sich um ein Boundary-Objekt handelt, gehört die neue Klasse in das Paket
devisor.framework.foundation.boundary und wird **BoundaryCone** genannt. Diese Klasse muß von **BoundaryBase** erben und das Interface **Segment** implementieren. Sie muß die zwei Punkte und den Radius verwalten können und die Methode **getInSphere**, die eine Kugel, die den Kegel umschließt, zurückgibt und deren Mittelpunkt die Mitte des Kegels ist, enthalten. Zusätzlich sollte sie auch die Methoden **getAbsolutePosition** und **getRelativePosition** implementieren, um das Setzen von Instanzen von **BoundaryNode3D** zu ermöglichen. Die Implementation von **BoundaryCylinder** sollte hier wiederum als Beispiel dienen.
Jetzt muß man die neue Klasse der Domain bekannt machen und sie entsprechend verlinken. Dazu modifiziere man die Klasse **Boundary** im gleichen Paket. Man implementiere eine neue Methode, **createBoundaryCone**, wobei man sich wiederum an der Methode **createBoundaryCylinder** ein Beispiel nehmen sollte. Vor allem ist hier der Aufruf von der Methode **addBoundary** wichtig, damit alle Verweise richtig gesetzt werden. Schon existiert intern eine neue Datenstruktur. Nun muß sie nur noch angezeigt werden.

B.4.3 Modifikation an den View-Klassen

Man erstellt einen MouseListener, der es ermöglicht, einen Kegel mit der Maus zu erstellen und einen Dialog, der es erlaubt, einen Kegel durch Eingabe der Eckdaten einzugeben.
Beiden gemeinsam ist die Zeichenroutine, die in der Klasse
devisor.framework.viewer.DrawingArea eingetragen werden muß. Zunächst muß die Methode drawBoundary so modifiziert werden, daß auch BoundaryCones an die neu zu implementie-

rende Zeichenroutine **drawCones** übergeben werden. Beim Zeichnen sollte man berücksichtigen, ob die Objekte gefüllt werden sollen, ob sie versteckt sind, und ob sie eventuell – durch die ViewBox eingeschränkt – nur teilweise sichtbar sind.

Eine neue Maus-Eingabe-Routine hinzufügen

Für eine Maus-Eingabe-Routine muß man eine eigene Ereignisbehandlungsklasse schreiben. Diese kommt in das Paket **devisor.grid.event** und wird z.B. **AddConeListener** genannt. Diese ist für Cone fast genauso wie die Klasse **AddCylinderListener** aufgebaut. Man muß die einzelnen Routinen für ein, zwei und drei Mausklicks modifizieren. Die Zeichenroutinen können, müssen aber nicht, sich von der Zeichenroutine in **DrawingArea** unterscheiden.
Nun muß der neue Listener in **devisor.grid.GridMainFrame** deklariert werden. Dazu muß ein neuer **MouseModeListener**, z.B. mit dem Namen **coneMouseListener** deklariert werden. Ein neues **JRadioButtonMenuItem** coneMenuItem sollte den Menüeintrag repräsentieren. Ein **JToggleButton** coneModeButton repräsentiert das neue Werkzeug in der Werkzeugleiste. In der Methode **createListeners** kommt dann hinzu (Codebeispiel 64):

```
coneModeListener =
    new MouseModeListener(this, GridManager.ADD_CONE);
```

*Codebeispiel 64: Modifikation an der Methode **create Listeners***

Vorher sollte man in der Klasse **devisor.grid.backend.GridManager** die neue Konstante **ADD_CONE** mit einem sinnvollen Wert eintragen.
In der Methode **GridMainFrame.createToolBar** sollte man analog zu Cylinder den Toggle-Button für Cone erstellen (Das Iconbild ’Cone.png’ sollte natürlich schon in **devisor.grid.images liegen)** (Codebeispiel 65):

```
    coneModeButton =
    new JToggleButton(new ImageIcon(getClass().getResource(
    imagepath + "Cone.png")));
    coneModeButton.addActionListener(coneModeListener);
    coneModeButton.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
            coneModeItem.setSelected(true); };
        });
    modeGroup.add(coneModeButton);
    editToolbar.add(coneModeButton);
    threeDComponentVector.addElement(coneModeButton);
    threeDComponentVector.addElement(coneButton);
```

*Codebeispiel 65: Modifikation in **GridMainFrame.createToolBar***

Ähnliche Modifikationen müssen analog zu Cylinder auch für Cone in der Methode **createMenu** durchgeführt werden. Auch in der Methode **changeMouseMode** muß einiges verändert werden (Codebeispiel 66):

```
    ...
    switch (mode) {
        ...
        case GridManager.ADD_CONE:
            setViewerCursor(crosshair_c);
            viewers[0].addMouseListener(
            oldml[0] =
            new AddConeListener(this,
                               (ViewerPanel2D)viewers[0]));
            viewers[1].addMouseListener(oldml[1] =
            new AddConeListener(this,
                               (ViewerPanel2D)viewers[1]));
            viewers[2].addMouseListener(oldml[2] =
            new AddConeListener(this,
                               (ViewerPanel2D)viewers[2]));
            viewers[0].addMouseMotionListener(oldml[0]);
            viewers[1].addMouseMotionListener(oldml[1]);
            viewers[2].addMouseMotionListener(oldml[2]);
            getStatusBar().
                  setText(gridi18n.
                  getString("AddConeHelp"));
            for (int i = 0; i < 3; i++)
                viewers[i].
                setInfo(gridi18n.
                getString("AddConeMode"));
            break;
        ...
```

Codebeispiel 66: Modifikation in ***GridMainFrame.changeMouseMode***

Einen neuen Dialog hinzufügen

Um einen neuen Dialog hinzuzufügen, erstellt man in dem Paket **devisor.grid.dialogs** eine Klasse, z.B. mit dem Namen **ConeDialog**. Sie wird analog zu der Klasse **CylinderDialog** erstellt. In diesem Fall könnte der Dialog sogar eins zu eins übernommen werden, nur die Verweise auf die Klassen der Domain müssen entsprechend modifiziert werden. In **devisor.grid.GridMainFrame** muß man wie beim **MouseListener** entsprechende Veränderungen vornehmen:

1. **coneDialog** deklarieren, in **createDialogs** initialisieren.
2. **coneButton** deklarieren und in **createToolBar** erstellen (analog zu CylinderButton)
3. Für **coneItem** in **createMenu** die gleiche Prozedur
4. Eine Zeile in **changeViewMode** im 2D-Fall hinzufügen (Codebeispiel 67):

```
    coneDialog.setVisible(false);
```

Codebeispiel 67: Modifikation in ***GridMainFrame.changeViewMode***

5. Die **update**-Routine muß analog zum **cylinderDialog** aktualisiert werden.

B.4.4 Weitere Modifikationen

Der Kegel wird korrekt gezeichnet und verwaltet. Allerdings könnten sich noch Probleme in anderen Dialogen ergeben. Beim **devisor.grid.dialogs.BoundaryDeleteDialog** und dem ähnlichen **devisor.grid.dialogs.VisibilityDialog** sollte die Methode **initDialog** analog zu Cylinder aktualisiert werden.
Auch das 3D-Viewer-Fenster kennt das neue Objekt noch nicht. Dafür müssen in **devisor.vision.grid** ensprechende Änderungen vorgenommen werden. Wie das geht, wird in der Vision-Beschreibung (s. Kap. 3.6) erklärt.
Genauso wie Cone, können beliebige Zeichenobjekte in Grid hinzugefügt werden.

B.5 Sonstige Erweiterungsmöglichkeiten

Folgende Erweiterungsmöglichkeiten sind noch offen:

B.5.1 Portierung nach Windows

Mit Ausnahme des Servers NEXUS ist der DEVISOR durch seine reine Java-Implementierung vollständig lauffähig unter allen Betriebssystemen, die die Java Laufzeitumgebung in der Version 1.4.1 zur Verfügung stellen. NEXUS ist allerdings ein reines ANSI-C Programm und muß somit für die jeweilige Architektur plattformabhängig übersetzt werden. Bisher basiert die Implementierung auf der Verwendung von UNIX-Sockets und ist somit auf Unix- und Linux-Derivate beschränkt. Die Erweiterung auf die Verwendung von WINSOCKS, der Windows-eigenen Implementierung der UNIX-Sockets wäre eine durchaus vorstellbare Erweiterung. Von der Projektgruppe wurde sie nicht vorgenommen, da die Zielplattform für den Server explizit auf Unix- bzw. Linux-Derivate festgesetzt wurde.
Eine Alternative ist die Verwendung von CygWin [30], wie für FEATFLOW in [21] beschrieben.Diese Alternative wurde ansatzweise ausprobiert, ist jedoch zum Zeitpunkt der Drucklegung dieses Berichts noch nicht vollständig ausgetestet.

B.5.2 Benutzerauthentifizierung

Im System ist bereits implementiert, die Datenzuordnung in Abhängigkeit von Benutzernamen zu gestalten bzw. den Zugriff auf bestimmte Daten vom Benutzernamen abhängig zu machen. Hier wäre eine Erweiterung um eine sichere, passwortgeschützte Benutzerauthentifizierung eine sinnvolle Erweiterung.
In diesem Zusammenhang wäre auch die Verschlüsselung der über das Netz transferierten Daten sinnvoll, da in der von der Projektgruppe implementierten Version alle Daten im Klartext übertragen werden.

Anhang C

API-Dokumentation

C.1 Übersicht über die Pakete und Klassen

In diesem Kapitel sollte der gesamte Code der DEVISOR-Module vorgestellt werden. An Modulen besitzt das DEVISOR-Paket die Module VISION, CONTROL, GRID, NET, und das Modul NUMERIK.
Da die komplette API-Dokumentation hier jeden Rahmen sprengen würde, wird zur Referenz auf die beiliegende CD verwiesen. Sie steht dort im HTML-Format zur Verfügung.

C.2 Das Paket CONTROL

Dieses Paket beinhaltet alle Klassen für das Modul CONTROL.

C.2.1 devisor.control.app

In diesem Paket werden die Server und Projekte verwaltet und es erfolgt die Anbindung an den **ControlWrapper**.

Control Dies ist die Hauptklasse des Pakets. Hier wird die Serverliste, die Projektliste und der **ControlWrapper** verwaltet.

DummyWrapper Ein Dummy, der einen **ControlWrapper** simuliert. Nur für Testzwecke.

ModuleException Exception für Fehler, die bei der Verwaltung des Moduls auftauchen.

ModuleObject In dieser Klasse wird ein Modul verwaltet und es erfolgt die Anbindung an den **ControlWrapper**.

ModuleObjectList Eine Liste, in der alle Module des gleichen Modultyps eines Projekts enthalten sind.

ProjectObject In dieser Klasse wird ein Projekt verwaltet.

ProjectObjectList Eine Liste, die alle in CONTROL aktiven Projekte enthält.

ResultListObject Wird für den Download benötigt.

Server In dieser Klasse wird ein Server verwaltet.

ServerList Eine Liste, die alle Server enthält.

ServerModulesTable Durch eine Hashtable wird die Serverliste verwaltet.

C.2.2 devisor.control.event

CloseTabEvent Wird gefeuert wenn eine JCloseableTabbedPane geschlossen werden soll.

CloseTabListener Hört auf CloseTabEvents.

ServerChangedEvent Wird bei einer Veränderung der Servereigenschaften gefeuert.

ServerChangedListener Hört auf ServerChangedEvents.

ServerListChangedEvent Wird bei einer Veränderung der Einträge der Serverliste gefeuert.

ServerListChangedListener Hört auf ServerListChangedEvents.

C.2.3 devisor.control.gui

DEVISOR Dies ist die Hauptklasse von CONTROL. Hiermit wird das Modul gestartet. Erzeugt auch das Hauptfenster.

JCheckConnectionDialog Bei Anlegen eines neuen Servers kann in diesem Dialog die Verbindung getestet werden.

JClockLabel Eine Uhr.

JCloseableTabbedPane Die TabbedPane für ein Modul.

JComponentCellRenderer Stellt beliebige JComponents in einer JTable dar.

JComponentTableCellEditor Erlaubt beliebige JComponents in einer JTable zu editieren.

JConvergenceBar Da läuft was.

JDeViSoRPropertiesDialog Ein Dialog, um Einstellungen des CONTROL-Moduls zu verändern, z.B. Hintergrundfarbe, Sprache,...

JDeViSoRSplashDialog Splashscreen für das Starten von CONTROL.

JDevisorAboutDialog Ein schöner Aboutdialog.

JErrorPanel In diesem Panel werden Statusmeldungen, Fehlermeldungen und Statistiken eines Moduls ausgegeben.

JModuleAdministrationPanel Ein Panel zur Steuerung eines Moduls.

JMultiLineToolTip Ermöglicht ToolTips mit mehreren Reihen.

JNewProjectDialog Ein Dialog zum Anlegen eines neuen Projekts.

JNewServerDialog Ein Dialog zum Anlegen eines neuen Servers.

JProjectInternalFrame Ein internes Fenster zur Verwaltung eines Projekts.

JProjectRadioButtonMenuItem Wird zum Umschalten der einzelnen Projektfenster benutzt.

JResultDownloadDialog Ein Dialog für den Download von Ergebnissen.

JResultTable Eine Tabelle zur Darstellung der Ergebnisse im Downloaddialog.

JServerList Ein Dialog für die Serverauswahl und -verwaltung.

ResultObjectTableModel Zur Verwaltung der Ergebnisse im Downloaddialog.

ProjectFileFilter Ein Filter für einen Standarddateiauswahldialog. Filtert Dateien vom Format „`*.devisor`“.

C.2.4 devisor.control.gui.dynamicGui

Dieses Paket enthält alle Klassen für die Erstellung der dynamischen GUI.

AbstractDynGuiPanel Abstrakte Oberklasse für alle Parameterpanels. Legt Methoden fest, die von allen Parameterpanels implementiert werden müssen. Diese Klasse ist gewissermaßen das Pendant zur Klasse **Parameter** aus dem Paket NET.

ExampleTree Erstellt eine **ParameterList** für Testzwecke. Diese enthält sortiert alle möglichen Parameter-GUI-Kombinationen.

JDataSourcePanel Ein Panel zur Konfiguration eines **DataSourceParameter**.

JDomainViewerDialog Dieser Dialog enthält ein Viewerpanel für Domains, die hiermit betrachtet werden können. Die Funktionalität beschränkt sich auf Zoomen und Verschieben der Domain.

JDynamicGuiDialog Erstellt für eine Parameterliste eine dynamische GUI.

JDynamicGuiPanel Dieses Panel enthält die dynamische GUI.

JFunctionPanel Ein Panel zur Konfiguration eines **FunctionParameter**.

JGridPanel Ein Panel zur Konfiguration eines **GridParameter**.

JModuleConfigurationDialog Der Hauptdialog für eine komplette Konfiguration.

JProblemConfigurationDialog Der Hauptdialog für die Konfiguration eines gegebenen Problems.

JScalarPanel Ein Panel zur Konfiguration eines **ScalarParameter**.

JSelectProblemPanel Ein Panel für die Auswahl eines Problems.

JSelectTimeStepsDialog Ein Dialog für die Auswahl von Zeitschritten.

JSelectionGuiDialog Ein Dialog für die Konfiguration von Parametern.

JSelectionPanel Ein Panel zur Konfiguration eines **SelectionParameter**.

JVectorPanel Ein Panel zur Konfiguration eines **VectorParameter**.

ParamTreeNode Verwaltet einen Knoten des Parameterbaums.

C.2.5 devisor.control.gui.properties

JAppearancePanel Ist das Panel zum Einstellen der Erscheinung von DEVISOR, also Sprache, Farbe des Desktops etc.

JPropertyPanel Ist das Interface welches von allen Panels des Properties-dialogs von DEVISOR implementiert werden muss.

C.2.6 devisor.control.gui.images

Enthält alle Bilder und Icons, die in der GUI verwendet werden.

C.2.7 devisor.control.util

DeViSoRProperties Singleton der die Einstellungen des DEVISOR kapselt.

DeViSoRRessourceBundle Singleton der die Übersetzungsdateien für DEVISOR kapselt.

Package Erlaubt das Suchen von Klassen und Klassennamen in einem Paket des Klassenpfades, welches lokal als Jar-Archiv oder entpackt vorliegt.

SimpleFileFilter Ist ein Filter für JFileChooser oder File und kann nach Dateiendungen oder nach Teilen des Namens, mit oder ohne Auflistung von Unterverzeichnissen filtern.

C.3 Das Paket FRAMEWORK

C.3.1 devisor.framework.foundation

Das **foundation**-Paket ist in mehrere Unterpakete aufgeteilt:

base Dieses Paket enthält die Klassen, auf denen alle weiteren Datenstrukturen aufbauen. Dies sind **Node**, **Triangle** und die InterfacesSegment und **Segment3D**.

boundary Dieses Paket enthält die Klassen, die Boundaries beschreiben. Oberklasse für alle Boundaries ist die Klasse **BoundaryBase**. Soll es möglich sein, einen **BoundaryNode** auf einer Boundary zu plazieren, so muss dieses das Interface **Segment** implementieren. Analog kann man einen **BoundaryNode3D** nur auf einem Boundary liegen, die **Segment3D** implementiert. Die Klasse **Boundary** ist ein Container für alle Boundaryklassen. Sie enthält Methoden, die die Beziehungen der enthaltenen Boundaries (z.B. die Liste der Eltern) auf dem aktuellen Stand hält.

domain Dieses Paket enthält die Klasse **Domain**. Sie dient als Container für alle Finiten Elemente (Paket devisor.grid.elements) und enthält außerdem eine Instanz der **Boundary**-Klasse und verwaltet somit alle Objekte, die eine Domain enthalten kann. Für jedes Objekt **XYZ** gibt es einen Methode **addXYZ** und **removeXYZ**, die Beziehungen der Objekte untereinander aktuell hält (z.B. die Lste der Eltern). Außerdem gibt es **createXYZ**-Methoden, die das Objekt **XYZ** erzeugen, falls gültige Parameter übergeben werden. Ist dies der Fall, so wird eine neue Instanz des Objektes mit einer eindeutigen Nummer zurückgeliefert. Ansonsten wird eine Exception erzeugt. Die entsprechenden Exceptions sind ebenfalls in diesem Paket enthalten.

elements Diese Paket enthält Klassen, die Finite Elemente beschreiben. Die einzelnen Klassen bauen aufeinander auf:

Eine **Edge** besteht aus zwei **Nodes**, ein **Quad** besteht aus vier **Edge**s, ein **Hexa** aus sechs **Quad**s, usw.

Dadurch sind auch die Nachbarschaftsbeziehungen zwischen den einzelnen Elementen festgelegt. Haben beispielsweise zwei Edges einen gemeinsamen Node, so sind diese benachbart.

Zusätzlich enthalten einige Objekte Informationen, wie sie automatisch weiter verfeinert werden können.

C.3.2 devisor.framework.i18n

In diesem Paket befinden sich die Dateien zur Internationalisierung. Im Moment sind dies die Sprachen Deutsch und Englisch.

C.3.3 devisor.framework.images

Dies ist nicht wirklich ein Paket, sondern lediglich das Verzeichnis, das die Bilder für die Icons enthält.

C.3.4 devisor.framework.mainframe

Das mainframe-Paket ist ein Überrest der Vorgängerversion von GRID und enthält nur noch einige Dialoge, die wiederverwendet werden konnten.

C.3.5 devisor.framework.options

Das **options**-Paket enthält die Optionsverwaltung, die aus der Vorgängerversion von GRID übernommen wurde. Der **OptionsManager** stellt eine Schnittstelle zur Verfügung, mit der man einheitlichen Zugriff auf die Optionsvariablen hat, und erlaubt es, die Daten persistent zu halten. Möchte man eigene Optionen verwalten, so muß man eine Klasse erstellen, die von **AbstractOptions** erbt.

C.3.6 devisor.framework.toolbox

Das Paket **toolbox** enthält einige Hilfsklassen. Die wichtigsten sind **BoundingBox** und **GeometryExtractor**.

Die Klasse **BoundingBox** erleichtert es, Boundingboxen von Objekten zu berechnen. Mit der Methode **add** können neue Punkte hinzugefügt werden. Mit **getUpperLeft** und **getLowerRight** kann man dann die Extremkoordinaten abfragen.
Die Klasse **GeometryExtractor** extrahiert eine Menge von Dreiecken (**GeometryTriangle**) aus einem Java3D-Szenegraphen.

C.3.7 devisor.framework.viewer

Das Paket **viewer** enthält Klassen, die die Schnittstelle zwischen den Foundationklassen und Java Swing bilden.

DomainOptions dient der Verwaltung der Anzeigeoptionen.

DrawingArea erbt von der Klasse **JPanel** und zeigt eine feste Ansicht der zugeordneten **Domain**-Klasse. Die Ebenen, die dargestellt werden können sind die XY-, die XZ- und die YZ-Ansicht.

MoveListener aktualisiert die Koordinaten, die in der ViewerPanel dargestellt werden.

Painter Klassen, die von Painter erben, können mit der Methode **addPainter** einer **DrawingArea** zugewiesen werden. Sie werden dann jedesmal aufgerufen, wenn diese aktualisiert wird. Mit **removePainter** können sie wieder entfernt werden.

ViewerPanel ist Elternklasse für alle Panels, die **Domain**s darstellen können.

ViewerPanel2D dient der Darstellung von **Domain**s in 2D. Jede Instanz enthält eine **DrawingArea**.

ViewerPanel3D dient der Darstellung einer **Domain** in 3D. Hierzu wird ein **devisor.vision.grid.DomainPanel** eingebunden.

C.4 Das Paket GRID

Dieses Paket beinhaltet Klassen zum Modul **devisor.grid**.

C.4.1 devisor.grid.backend

In diesem Paket befinden sich die Klassen **GridManager** und **StatusBar**. `GridManager` ist eine Klasse, die verschiedene, in **devisor.grid.GridMainFrame** gebrauchten Konstanten und Mouse-Modis verwaltet. `StatusBar` repräsentiert einen Fortschrittsbalken, der z.B. beim Laden einer Domain angezeigt wird.

C.4.2 devisor.grid.dialogs

Dieses Paket beinhaltet alle Dialoge des Moduls GRID. Die meisten Dialoge dienen der Eingabe von Zeichenelementen.

AboutDialog Dieser Dialog zeigt die Entwickler, die GPL-Lizenz und weitere Informationen über das GRID-Projekt.

BoundaryCubicleDialog Dieser Dialog dient als Eingabemöglichkeit für BoundaryCubicles.

BoundaryDeleteDialog Dieser Dialog dient zum Löschen von beliebigen Boundary-Objekten.

BoundaryNode3DDialog Dieser Dialog dient als Eingabemöglichkeit für BoundaryNode3D's.

BoundaryNodeDialog Dieser Dialog dient als Eingabemöglichkeit für BoundaryNodes.

CircleDialog Dieser Dialog dient als Eingabemöglichkeit für Circles.

BoundaryCylinderDialog Dieser Dialog dient als Eingabemöglichkeit für BoundaryCylinder.

EdgeDialog Dieser Dialog dient als Eingabemöglichkeit für Edges.

ExceptionDialog Dieser Dialog wird bei einer Ausnahme eingeblendet. Der Text der Ausnahme wird darin vollständig aufgeführt.

GridSplashDialog Dieser Dialog wird beim Start von GRID gezeigt. Ein Fortschrittsbalken visualisiert den Ladevorgang

HelpDialog Dieser Dialog zeigt das GRID-Handbuch (Kap. A.5) im HTML-Format.

HexaDialog Dieser Dialog dient als Eingabemöglichkeit für Hexa-Objekte.

HierarchyDialog Dieser Dialog zeigt den hierarchischen Aufbau der aktuell bearbeiteten Domain mit all ihren Elementen in einer Verzeichnis-ähnlichen Struktur.

NodeDialog Dieser Dialog dient als Eingabemöglichkeit für Nodes.

OptionsDialog Dieser Dialog dient als Eingabemöglichkeit für Optionen des Programms.

PasteAtDialog Dieser Dialog dient als Eingabemöglichkeit für die Stelle auf der zeichenfläche, an die die kopierten Elemente eingefügt werden sollen.

PointOfInterestDialog Dieser Dialog dient als Eingabemöglichkeit für die Mittelpunkte der Arbeitsfenster Top, Side und Front.

QuadDialog Dieser Dialog dient als Eingabemöglichkeit für Quad-Objekte.

SphereDialog Dieser Dialog dient als Eingabemöglichkeit für BoundarySpheres.

TetraDialog Dieser Dialog dient als Eingabemöglichkeit für Tetra-Objekte.

TriDialog Dieser Dialog dient als Eingabemöglichkeit für Tris-Objekte.

ViewBoxDialog Dieser Dialog dient als Eingabemöglichkeit für die Anzeigebeschränkung in den Arbeitsfenstern.

VisibilityDialog In diesem Dialog kann man einstellen, welche Boundary-Objekte angezeigt werden und welche nicht. item[HexaDialog] Dieser Dialog dient als Eingabemöglichkeit für Hexa-Objekte.

C.4.3 devisor.grid.event

In diesem Packet befinden sich ausschließlich Klassen, die die Mouse-Ereignisbehandlung in GRID realisieren

AddCircleListener Dieser Listener behandelt die Ereignisse beim Erstellen eines Circles.

AddCubicleListener Dieser Listener behandelt die Ereignisse beim Erstellen eines BoundaryCubicles.

AddCylinderListener Dieser Listener behandelt die Ereignisse beim Erstellen eines BoundaryCylinders.

AddEdgeListener Dieser Listener behandelt die Ereignisse beim Erstellen einer Edge.

AddNodeListener Dieser Listener behandelt die Ereignisse beim Erstellen eines Node.

AddQuadListener Dieser Listener behandelt die Ereignisse beim Erstellen eines Quads.

AddSegmentListListener Dieser Listener behandelt die Ereignisse beim Erstellen einer BoundarySegmentList.

AddSphereListener Dieser Listener behandelt die Ereignisse beim Erstellen einer BoundarySphere.

AddTriListener Dieser Listener behandelt die Ereignisse beim Erstellen eines Tri-Objekts.

DeleteSelectionListener Dieser Listener behandelt die Ereignisse beim Löschen der Auswahl.

ExitListener Dieser Listener behandelt die Ereignisse beim Beenden von GRID.

GridKeyListener Dieser Listener behandelt die Ereignisse bei der Tastatureingabe.

MoveSelectionListener Dieser Listener behandelt die Ereignisse beim Bewegen der Auswahl.

MoveViewListener Dieser Listener behandelt die Ereignisse beim Bewegen der Ansicht eines Fensters.

NewProjectListener Dieser Listener behandelt die Ereignisse beim Erstellen einer neuen Domain.

RotateSelectionListener Dieser Listener behandelt die Ereignisse beim Rotieren der Auswahl.

ScaleSelectionListener Dieser Listener behandelt die Ereignisse beim Skalieren der Auswahl.

SelectBoxListener Dieser Listener behandelt die Ereignisse beim Auswählen der Zeichenelemente.

ShowDialogListener Dieser Listener behandelt die Ereignisse beim Anzeigen der Dialoge.

ShowHierarchyListener Dieser Listener behandelt die Ereignisse beim Anzeigen des Hierarchiefensters.

ShowPointOfInterestDialogListener Dieser Listener behandelt die Ereignisse beim Anzeigen des PointOfInterest-Dialogs.

TriangulationListener Dieser Listener behandelt die Ereignisse beim Erstellen einer BoundaryTriangulation (Dieser Listener wird erst in der nächsten Version von GRID verfügbar sein).

ViewBoxDialogListener Dieser Listener behandelt die Ereignisse beim Anzeigen eines ViewBoxDialogs.

ZoomFactorListener Dieser Listener behandelt die Ereignisse beim Einstellen des Zoomfaktors.

C.4.4 devisor.grid.i18n

Dieses Paket beinhaltet keine Klassen, sondern die Lokalisierungsdateien **gridi18n.***. Mit diesen wird die Mehrsprach-Fähigkeit von GRID gewährleistet.

C.4.5 devisor.grid.images

Dieses Paket enthält ein paar Bilder, die in GRID verwendet werden. Weitere Bilder befinden sich in **devisor.framework.images**

C.4.6 devisor.grid.info

Dieses Paket beinhaltet alle Dateien, die zu den Hilfe-Menüpunkten ”?” in GRID gehören.

C.4.7 devisor.grid.main

Dieses Paket ist das wichtigste Paket, denn hier befinden sich die Kernklassen von GRID.

FeastLoader Diese Klasse verwaltet den Ladevorgang beim Laden einer FEAST-Datei in die Domain.

GridApp Mit dieser Klasse wird GRID gestartet. Es werden dabei die gespeicherten Optionen geladen oder die hart kodierten übernommen.

GridMainFrame Diese Klasse stellt das Hauptfenster von GRID dar. Gleichzeitig Verwaltet sie die Domain, alle Dialoge und alle Listener.

GridNet Diese Klasse ist eine Verbindungsklasse zu der Klasse **devisor.net.wrappers.GridWrapper**, mit der dann die Anbindung ans Netz und dadurch an das DEVISOR-Paket vollzogen wird.

GridOptions Diese Klasse verwaltet die Mouse-Modi des GRID-Hauptfensters.

MouseModeListener Diese Klasse repräsentiert einen Mouse-Modus des Hauptfensters.

WaveFrontImporter Diese Klasse importiert Java Wave-Front Dateien in die Domain.

C.4.8 devisor.grid.options

Dieses Paket enthält nur eine einzige Klasse, die **GridLocale_en_US**, die die Lokalisierung innerhalb von GRID verwaltet.

C.5 Das Paket NET

Dieses Paket beinhaltet Klassen zur Netzwerkfunktionalität. Es besteht aus der Klasse **NetException** und den Unterpaketen **wrappers**, **interim**, **example** und **debug**.

C.5.1 devisor.net

In diesem Paket befindet sich lediglich die Klasse **NetException**, da sie logisch keinem anderen Paket zuzuordnen ist. Sie dient als Container für alle Exceptions, die innerhalb des Pakets auftreten können und an andere Module weitergegeben werden. Dazu kapselt sie Informationen, die sie eindeutig einer Quelle zuordnen lassen, sowie eine Liste (in Form einer **ListOfErrors**, siehe Kap. C.5.3) von Fehlermeldungen. Diese Klasse kann komplett über ihren Konstruktor verwendet werden.

C.5.2 devisor.net.wrappers

Dieses Paket beinhaltet die zentralen Schnittstellen zwischen Modulen und dem Netzwerk.

ControlWrapper Diese Klasse stellt für jeden Protokollbefehl (s. Kap. D.3) eine Methode zur Verfügung, die eben diese Protokollsequenz mit dem CONTROL-Modul als Sender durchführt. Außerdem wird eine Art *Posteingang* angeboten, in dem Huckepack-Nachrichten (s. Kap. 3.2.4) abgelegt werden.

ModuleLock Diese Klasse implementiert einen einfachen lock-Mechanismus und macht so den **ControlWrapper** threadsafe.

NotificationQueue Diese Klasse kapselt eben diesen Posteingang in Form einer FIFO-Warteschlange. Es können Objekte vom Typ **Statistics** und **ListOfErrors** (s. Kap. C.5.3) eingefügt werden. Sie ist für den *single consumer multiple producers*-Einsatz synchronisiert.

KeepAliveThread Diese Klasse kapselt ein ping-Signal für bestehende Verbindungen. Zusätzlich zur einfachen ping-Antwort können als Antwort Objekte vom Typ **Statistics** und **ListOfErrors** (s. Kap. C.5.3) empfangen werden.

AbstractModuleWrapper Dies ist die abstrakte Superklasse aller Modulwrapper mit Ausnahme des CONTROL-Moduls. Ankommende Verbindungen werden akzeptiert und an eine neu gestartete Instanz des **AbstractCommunicationThread** (s.u.) weitergeleitet.

Wrappable Ein Interface, das von allen Modulen implementiert werden muß, um mit dem **AbstractModuleWrapper** und **AbstractCommunicationThread** kommunizieren zu können. Es definiert Methoden, um den Nachrichtenaustausch zwischen Wrapper und Modul sicherzustellen.

AbstractCommunicationThread Diese Klasse kapselt eine komplette Kommunikationssequenz (Parsen der ankommenden Nachricht, Generierung der Antwort und deren Rücksendung). Insbesondere ist das vollständige Protokoll für ein von CONTROL verschiedenes Modul implementiert.

GridWrapper Dies ist die speziell an die Bedürfnisse des Moduls GRID angepaßte Erweiterung des **AbstractModuleWrapper**.

GridCommunicationThread Die speziell an die Bedürfnisse des GRID-Moduls angepaßte Erweiterung des **AbstractCommunicationThread**. Für das Modul irrelevante Protokollbefehle werden mit einer Fehlermeldung quittiert.

VisionWrapper Eine speziell an die Bedürfnisse des Moduls VISION angepaßte Erweiterung des **AbstractModuleWrapper**.

VisionCommunicationThread Die speziell an die Bedürfnisse des VISION-Moduls angepaßte Erweiterung des **AbstractCommunicationThread**.

WrapperUtilities Diese Klasse bietet einige einfache statische Methoden, die von den Klassen in diesem Paket häufig benötigt werden.

C.5.3 devisor.net.interim

Dieses Paket definiert alle notwendigen Containerklassen und ihre Parser, um Daten über die Klassen aus dem **wrappers**-Paket (s. Kap. C.5.2) zwischen den Modulen hin- und herzusenden.

Parsable Dies ist das wichtigste Interface des gesamten Pakets und schreibt zwei Methoden vor, um den Inhalt einer Containerklasse in ihre Protokollrepräsentation umzuwandeln und umgekehrt.

Header, Footer Diese Klassen kapseln den Header und den Footer jeder Protokollsequenz.

Error Diese Klasse kapselt (pessimistischerweise) einfache Textnachrichten (d.h. Fehler- und Statusmeldungen), die versendet werden sollen. Zusätzlich können sie durch diverse Zusatzangaben eindeutig einer Quelle zugeordnet werden.

ListOfErrors Eine einfache Listenumgebung für Objekte vom Typ **Error**.

Module Diese Klasse kapselt die zur Beschreibung eines Moduls nötigen Informationen. Außerdem dient sie als Übergabeparameter für alle Methoden der Klasse **ControlWrapper** (s. Kap. C.5.2) und beinhaltet dazu die notwendigen Netzwerk-Attribute wie Adresse und Port des Moduls, eingehende und ausgehende Datenströme usw.

ModuleList Eine Listenumgebung für Instanzen der Klasse **Module**.

Problem Diese Klasse kapselt eine Problembeschreibung mit allen dazu notwendigen Informationen.

ProblemList Eine Listenumgebung für Objekte vom Typ **Problem**.

Parameter Dies ist die abstrakte Oberklasse sämtlicher Parameterobjekte. Sie stellt allen Parametern gemeinsame Methoden und Attribute bereit.

GUIDescription Eines dieser gemeinsamen Attribute ist die Beschreibung, wie eine Instanz von **Parameter** in einer GUI dargestellt werden soll. **GUIDescription** ist die abstrakte Oberklasse all dieser Beschreibungen.

NodeParameter Eine Erweiterung von Parameter, um hierarchische Verzweigungen in Parameterlisten zu ermöglichen. Sie enthält dazu als Attribut eine weitere Parameterliste.

ScalarParameter Eine Erweiterung von **Parameter** zur Repräsentation von "skalaren" Daten wie Zahlwerten, Zeichenketten und boolschen Werten.

VektorParameter, Entry, EntryList Diese Klasse erweitert ebenfalls die abstrakte Parameterklasse und kapselt Objekte vom Typ **Entry** in einer Instanz von **EntryList**, denen eine gemeinsame GUI-Beschreibung zugeordnet wird. So lassen sich beispielsweise Comboboxen mit mehreren Parametern füllen.

GridParameter Diese **Parameter**-Erweiterung kapselt eine Gitter- und Geometriebeschreibung und bietet Konvertierungsroutinen zwischen der Klasse **Domain** (s. Kap. C.3) und dem FEAST-Format.

SelectionParameter Eine Parameter-Erweiterung, die eine n aus m-Auswahlmöglichkeit aus verschiedenen gegebenen **Parameter**-Instanzen implementiert.

FunctionParameter, SubParameter Diese ebenfalls von **Parameter** abgeleitete Klasse kapselt eine mathematische Funktion. Sie ist gegeben durch eine Menge an Übergabeparametern, einer Menge an intern in der Funktionsvorschrift deklarierten Parametern (gegeben in Instanzen der Klasse **SubParameter**) und der Berechnungsvorschrift selbst. Details zur Formatspezifikation finden sich in der Protokollspezifikation (s. Kap. D.3)

DataSourceParameter Diese Klasse kapselt einen der Parameter mit fester ID, der alle Informationen beinhaltet, die nötig sind, um ein Modul so zu konfigurieren, daß es ein anderes Modul als Datenquelle verwendet.

XYZDescription Die Klassen **CheckBoxDescription, ColorChooserDescription, ComboBoxDescription, FileChooserDescription, GridViewerDescription, NodeDescription, SelectionDescription, SliderDescription, TextFieldDescription, TextAreaDescription** erweitern **GUIDescription** um Beschreibungen für Checkboxen, Farbauswahldialoge, Comboboxen, Dateidialoge, eine reduzierte Ansicht eines 3D-Gitters, einer hierarchischen Verzweigung, eines n aus m Auswahldialogs, eines Sliders, und für ein- und mehrzeilige Textfelder.

RadioButtonDescription, Button, ButtonList Eine Erweiterung von **GUIDescription** zur Definition eines Feldes aus Radiobuttons. Die einzelnen Schaltflächen selbst werden durch Instanzen der Klasse **Button**, gesammelt in einer **ButtonList**.

ModuleCapabilities Diese Klasse dient als einfacher Container und stellt alle Informationen zur Verfügung, die ein Modul anbietet und benötigt: eine Liste der unterstützten Probleme, zu jedem Problem eine Liste der benötigten Parameter, und eine Liste der globalen – problemunabhängigen – Konfigurationsmöglichkeiten.

CompleteConfiguration Ein einfacher Container, um alle Informationen, die zur Konfiguration eines Moduls mit einem Problem nötig sind, zusammen verwalten zu können. Konkret handelt es sich um das zu lösende Problem, die gesetzten Parameter und die globale Konfigurationsliste.

Statistics Ein Statistik-Objekt kapselt alle zu einem Zeitschritt gehörenden Statistikdaten. Die Daten selbst stehen in Form einer Referenz auf eine **ParameterList**, gefüllt mit Instanzen der Klasse **ScalarParameter** zur Verfügung.

StatisticsList Eine einfache Listenumgebung für obige Objekte.

Result, TimeStep Instanzen dieser Klasse kapseln Ergebnisdaten, oder konkreter, Referenzen auf vorhandene Ergebnisdateien. Ein **Result** kann zunächst mehrere Zeitschritte umfassen, die zu einem Zeitschritt gehörenden Informationen werden dann in Referenzen auf Instanzen von **TimeStep** gespeichert.

ResultList Eine einfache Listenumgebung für Ergebnisse.

C.5.4 devisor.net.example

Dieses Paket enthält ein vollständig implementiertes Beispielmodul, das auch Kernkomponente des Tutorials zur Integration eines neuen Moduls ist (s. Kap. B.2).

ExampleModule Diese Klasse kapselt das Beispielmodul und implementiert seine – zugegebenermaßen nicht wirklich sinnvolle – Funktionalität.

ExampleWrapper Diese Klasse erweitert den **AbstractModuleWrapper** aus dem **devisor.-net.wrappers**-Paket um die Funktionen, die für das Beispielmodul nötig sind.

ExampleCommunicationThread Eine Erweiterung der abstrakten Klasse **AbstractCommunicationThread**.

ExampleMC Aus didaktischen Gründen wurde der Aufbau der nötigen Parameterlisten in diese isolierte Klasse verlagert.

C.5.5 devisor.net.debug

Dieses Paket beinhaltet einige Testklassen, um das Netzwerkpaket isoliert von graphischen Benutzeroberflächen automatisch testen zu können. Außerdem dienen sie als Beispiel für die Programmierbarbeit des Systems:

ControlProto Eine kleine Klasse mit graphischem Frontend, mit der Protokollbefehle aus Dateien geladen werden können und an einen Server übermittelt werden können. Sie dient zum interaktiven Testen des Servers.

ControlTest Die ausführliche Beispielklasse zur Programmierbarbeit. Der Name ist etwas irreführend, getestet wird das Netzwerk aus der Sicht des CONTROL-Moduls bzw. unter Verwendung des **ControlWrapper**.

GridTest, VisionTest Diese Klassen simulieren den automatischen Start und die Konfiguration der Module GRID und VISION.

C.6 Das Paket NEXUS

In diesem Unterkapitel werden die Routinen des Linkmoduls mit den Parameterlisten dargestellt, die die Kommunikation zum Serverprozeß realisieren. Die Bindungs haben einen Unterstrich im Namen, der in C-Programmen auch angegeben werden muß, während der Unterstrich in Fortran-Programmen weggelassen werden muß.

C.6.1 API-Dokumentation

```
void dllm_result_available_()
```

Codebeispiel 68: Befehl zur Ergebnisfreigabe

Nach Aufruf dieser Routine wird dem aufrufenden Server signalisiert, daß Resultate zur Verfügung stehen.

```
void dllm_is_running_(int* iret)

void dllm_is_paused_(int* iret)
```

Codebeispiel 69: Befehle zur Abfrage des Status

Die Routine liefert in `iret` `TRUE`, falls das Modul gestartet wurde, bzw. falls das Modul mit dem Pause-Befehl angehalten wurde, ansonsten `FALSE`.

```
void dllm_init_relink_(int* ih ,int* uis,
  int* uirs, int* ret)
```

Codebeispiel 70: Initialisierungsbefehl

Diese Routine reinitialisiert eine TCP-Verbidnung, die vorher von einem VINCULUM-Modul initiiert wurde. Das Instancehandle `ih`, das Sockethandle `uis` und das Remotesockethandle `uirs` werden dem Modul über die Kommandozeile übermittelt und müssen der Routine übergeben werden. In `ret` wird das Ergebnis des Befehls zurückgegeben.

```
void dllm_init_link_(int* ih, int* ret,int* ia)
```

Codebeispiel 71: Initialisierungsbefehl

Diese Routine initiiert die initiale Verbindung zum Servermodul. Ihr wird das Instancehandle `ih` übergeben. Zurückgegeben werden die Statusvariable `ret` und die Socketkennung `ia`.

```
void dllm_listen_return_(int* ret,int* ianswer)
```

Codebeispiel 72: Allgemeiner Rückmeldebefehl

Diese Routine liefert an den aufrufenden Server das Ergebnis zum übermittelten Kommando `ret` zurück. Der Fehlercode wird in `ianswer` gesetzt. Die verfügbaren Fehlercodes sind in der Datei `deverror.h` zu finden.

```
void dllm_return_feat_errcode_(int* ival)
```

Codebeispiel 73: Rückmeldebefehl für FEAT-Fehlercodes

Diese Routine übersetzt einen Fehlercode der Softwarebibliothek FEAT in den entsprechende DeViSoR-Fehlercode und ruft dann `dllm_listen_return_` auf.

```
void dllm_listen_link_(int* timestep, int* ret,
  int* value, int* retint, double* retdouble)
```

Codebeispiel 74: Abfrageroutine

Die Routine lauscht auf dem lokalen Port und gibt, falls eine Nachricht vorliegt, den Befehlscode in `ret` zurück, mit den Parametern `value`, `retint` und `retdouble`. Liegt keine Nachricht vor, wird in `ret` der Wert `LOCNOCMD` zurückgeliefet. Das Programm sollte dann nichts machen, bzw. mit dem letzten Kommando weitermachen. Die Routine erwartet in `timestep` die Nummer des aktuellen Zeitschrittes.

```
void dllm_link_exit_()
```

Codebeispiel 75: Endebefehl

Diese Routine schließt die lokale Verbindung und sollte vor dem Beenden des Programms aufgerufen werden.

```
void dllm_link_wait_()
```

Codebeispiel 76: Wartebefehl

Diese Routine wartet eine Sekunde.

```
void dllm_open_stat_(int* itime)

void dllm_close_stat_()
```

Codebeispiel 77: Statistikbefehle

Diese Routine öffnet bzw. schließt die zum Zeitschritt `itime` gehörige Statistikdatei.

```
void dllm_put_stat_int_(int* value)

void dllm_put_stat_double_(double* value)
```

Codebeispiel 78: Statistikbefehle

Die Routine schreibt den Integer-Statistikwert `value`, rsp. den Double-Statistikwert `value`, in die mittels `dllm_open_stat` geöffnete Statistikdatei. Die Sequenz der `dllm_put_-stat`-Befehle muß mit der in der Parameterdatei definierten übereinstimmen.

```
int dllm_getresult(char* mask,char* pid, char* user,
       int mid, int ih, int rid, int nots, int* ts,
       char* servername, int port)
```

Codebeispiel 79: Befehl zum Ergebnisdownload

Die Routine holt vom Server, welcher durch den Servernamen `servername` und den Port `port` spezifiziert ist, die durch Module-ID `mid`, Problem-ID `pid`, Username `user` und Result-ID `rid` definierten Ergebnisdateien. Die gewünschten Zeitschritte werden im Integerarray `ts` übergeben, die Anzahl der Zeitschritte in `nots`. Die Dateien werden mit dem durch `mask` definierten Namen und einer fortlaufenden Nummer abgespeichert. Im Fehlerfall gibt die Routine einen Fehlercode zurück.

```
#define LOCUTUS_NOCMD 1
#define LOCUTUS_OK 0
#define LOCUTUS_READDATA 2
#define LOCUTUS_START 3
#define LOCUTUS_STOP 4
#define LOCUTUS_STEP 5
#define LOCUTUS_FORWARD 6
#define LOCUTUS_REWIND 7
#define LOCUTUS_EXIT 8
#define LOCUTUS_UNKCMD 9
#define LOCUTUS_PAUSE 10
#define LOCUTUS_PASS 11
#define LOCUTUS_GETSTATUS 12
#define LOCUTUS_SETINTPAR 13
#define LOCUTUS_SETDOUBLEPAR 14
```

Codebeispiel 80: Befehlskonstanten

```
#define LOCUTUS_STATUS_UNKNOWN 0
#define LOCUTUS_STATUS_STARTED 1
#define LOCUTUS_STATUS_CONFIGURED 2
#define LOCUTUS_STATUS_RUNNING 3
#define LOCUTUS_STATUS_FINISHED 4
```

Codebeispiel 81: Modul-Status

```
#define LOCUTUS_TAPE_UNKNOWN 0
#define LOCUTUS_TAPE_PAUSE 1
#define LOCUTUS_TAPE_PLAY 2
#define LOCUTUS_TAPE_STOP 3
```

Codebeispiel 82: Modul-Kassettenrecorder-Status

Befehl	Bedeutung	ivalue	ipar	dpar	Bemerkung
NOCMD	kein Kommando	-	-	-	ignorieren
OK	Statusbefehl	-	-	-	ignorieren
PASS	Statusbefehl	-	-	-	ignorieren
GETSTATUS	Statusbefehl	-	-	-	ignorieren, wird intern bearbeitet
READDATA	Konfiguration	-	-	-	wird gewöhnlich von VINCULUM bearbeitet
START	Rechnung gestartet	-	-	-	-
STOP	Rechnung gestoppt	-	-	-	-
STEP	Rechnung gesteppt	-	-	-	-
FORWARD	Rechnung vorwärts gespult	Schrittzahl	-	-	-
REWIND	Rechnung rückwärts gespult	Schrittzahl	-	-	-
EXIT	Modul verlassen	-	-	-	-
UNKCMD	Unbekanntes Kommando	-	-	-	ignorieren
PAUSE	Rechnung soll pausieren	-	-	-	PAUSE hebt ein voriges PAUSE wieder auf
SETINTPAR	Integerparameter setzen	ID	Wert	-	-
SETDOUBLEPAR	Doubleparameter setzen	ID	Wert	-	

Tabelle C.1: Bedeutungen der Befehle und Parameter

C.7 Das Paket VISION

Dieses Paket beinhaltet Klassen zur Visualisierung der Simualtionsergebnisse. Die Hauptklasse PipeInstance implementiert das **Wrappable**-Interface des NET-Moduls und steuert den Ablauf eines Visualisierungsdurchgangs. Nähere Informationen finden sich im Kapitel Systembeschreibung.

C.7.1 devisor.vision

ResultCollector Diese Klasse dient dem Zusammentragen von von Vision erzeugten Bildern und Filmen, die über CONTROL heruntergeladen werden können.

I_Configurable Ein Interface, daß von allen Klassen, die zur Visualisierung verwendet werden,implementiert werden muß. Es schreibt Methoden zum Abfragen der Parameterlisten und zum Konfigurieren der Pipeline-Elemente vor.

C.7.2 devisor.vision.ds

DomainTransformer Erzeugt nicht-realistische GMV-Daten zu jeder beliebigen Domain. Für Testzwecke.

FineGrid Diese Klasse stellt eine Datenstruktur für ein verfeinertes Gitter zur Verfügung.

GMVCell Diese Klasse stellt eine Datenstruktur für eine Zelle im Gitter zur Verfügung.

GMVControl Diese Klasse stellt Methoden zur Validierung von GMV-Dateien zur Verfügung, nützlich für Testzwecke.

Testdomain Eine hartcodierte Domain zum Testen der Pipeline-Komponenten.

ValueNode Erweiterung der **Node**-Klasse aus dem **foundation**-Paket um vektorielle und skalare Werte an den Knoten.

VisionDomain Erweiterung der **Domain**-Klasse aus dem **foundation**-Paket um **ValueNodes**.

VisionEdge Erweiterung der **Edge**-Klasse aus dem **foundation**-Paket um **ValueNodes**.

VisionFace Erweiterung der **Face**-Klasse aus dem **foundation**-Paket um **ValueNodes**.

VisionHexa Erweiterung der **Hexa**-Klasse aus dem **foundation**-Paket um **ValueNodes**.

VisionQuad Erweiterung der **Quad**-Klasse aus dem **foundation**-Paket um **ValueNodes**.

VisionTetra Erweiterung der **Tetra**-Klasse aus dem **foundation**-Paket um **ValueNodes**.

VisionTri Erweiterung der **Tri**-Klasse aus dem **foundation**-Paket um **ValueNodes**.

C.7.3 devisor.vision.exceptions

ConfigurableException Diese **Exception** wird bei Problemen beim Einsammeln der Parameterlisten der Pipeline-Elemente oder beim Konfigurieren der Pipeline-Elemente benutzt.

ElementNotFoundException Dient zur Signalisierung von Problemen mit Elementen der **VisionDomain**.

ExceptionHandler Die zentrale Klasse zur Verwaltung von **Exception**s.

ExceptionHandlerException Wird benutzt, um einen nicht initialisierten **ExceptionHandler** zu signalisieren.

FilterException Signalisiert Fehler beim Filtervorgang.

HexaNotFoundException Signalisiert Fehler beim Zugriff auf ein Hexa.

MovieException Signalisiert Fehler beim Erzeugen eines Films.

QuadNotFoundException Signalisiert Fehler beim Zugriff auf ein Quad.

TetraNotFoundException Signalisiert Fehler beim Zugriff auf ein Tetra.

TriNotFoundException Signalisiert Fehler beim Zugriff auf ein Tri.

VisionException Signalisiert nicht näher spezifizierte Fehler bei der Visualisierung.

C.7.4 devisor.vision.grid

DomainPanel Diese Klasse kapselt eine vereinfachte Visualisierungs-Pipeline in eine Unterklasse von **JPanel**. Sie wird im Grid-Editor zur dreidimensionalen Visualisierung der erstellen Objekte verwendet.

C.7.5 devisor.vision.gui

VisionViewer Diese Klasse dient zum Anzeigen des von einem Renderer erzeugten **Canvas3D** sowie einer erklrärenden Legende auf dem Bildschirm.

C.7.6 devisor.vision.movie

MovieMaker Diese Klasse dient zum Erzeugen eines Films aus den erzeugten JPEG-Bildern.

MovieMakerGUI Diese Klasse dient dem einfachen Zugriff auf die MovieMaker-Funktionalität. Wird im normalen Visualisierungs-Vorgang nicht benutzt.

PlayMedia Diese Klasse dient dem einfachen Abspielen von Quicktime-Filmen. Wird im normalen Visualisierungsvorgang nicht benutzt.

SnapshotTaker Diese Klasse dient der Erzeugung von JPEG-Bildern aus den berechneten Ergebnissen der Pipeline.

C.7.7 devisor.vision.particle

DiffEqu Diese Klasse ist eine Datenstruktur für Differentialgleichungen.

DiffEquHelper Diese Klasse stellt Hilfsfunktionen für **DiffEqu** zur Verfügung.

ParticleFunction Diese Klasse stellt Grundfunktionen der Partikel-Verfolgung zur Verfügung.

ParticleFunction2d Diese Klasse stellt Grundfunktionen der zweidimensionalen Partikel-Verfolgung zur Verfügung. Nur für Testzwecke.

ParticleFunction3d Diese Klasse stellt Grundfunktionen der dreidimensionalen Partikel-Verfolgung zur Verfügung. Nur für Testzwecke.

ParticleFunctionHexa Diese Klasse stellt Grundfunktionen der Partikel-Verfolgung in Hexa-Domains zur Verfügung. Nur für Testzwecke.

ParticleTracing2D Diese Klasse führt eine zweidimensionale Partikel-Verfolgung durch.

ParticleTracing3D Diese Klasse führt eine dreidimensionale Partikel-Verfolgung durch.

RungeKutta Diese Klasse kapselt den Algorithmus von Runge-Kutta vierter Ordnung.

C.7.8 devisor.vision.pipe

PipeInstance Dies ist die zentrale Klasse der Visualisierungs-Pipeline. Bei jedem Start des VISION-Moduls wird eine Instanz erzeugt. Sie kümmert sich um das Zusammentragen der benötigten Parameter mittels der Klasse **FindConfigurables** aus dem **devisor.vision.util**-Pakets. Diese Informationen werden anschliessend an CONTROL übertragen, dort verändert und zur Konfiguration der Pipeline zurückgeschickt. Eine intensive Abhandlung zu dieser Klasse findet sich in der Systembeschreibung.

C.7.9 devisor.vision.pipe.filter

AbstractFilter Fasst einige allgemeine Filter-Funktionen zusammen und bietet somit eine Grundlage für Filter-Implementierungen. Implementiert die **I_Mapper**- und **I_Configurable**-Interfaces

CutlinesFilter Verkleinert eine zweidimensionale Domain um eine Dimension.

CutsurfacesFilter Verkleinert eine dreidimensionale Domain um eine Dimension.

I_Filter Schreibt Methoden vor, die von Filter-Implementierungen angeboten werden müssen.

IdentityFilter Läßt die Domain wie sie ist.

IsoLinesFilter Erzeugt aus einer Domain eine Anzahl von Iso-Linien.

ScalarThresholdFilter Schneidet die Werte in einer Domain an Grenzwerten ab.

Surface3DFilter Lenkt eine zweidimensionale Domain anhand der Knotenwerte in die dritte Dimension aus.

C.7.10 devisor.vision.pipe.mapper

AbstractArrowMapper Fasst einige allgemeine Funktionen für den **Arrow2D**- und den **Arrow3DMapper** zusammen.

AbstractGlyphMapper Fasst einige allgemeine Funktionen für die **AbstractArrow**- und den **AbstractSphere**-Klassen zusammen. und den Arrow3DMapper zusammen.

AbstractMapper Fasst einige allgemeine Mapper-Funktionen zusammen und bietet somit eine Grundlage für Maper-Implementierungen. Implementiert die **I_Mapper**- und **I_Con-fi-gu-rab-le**-Interfaces.

AbstractSphereMapper Fasst einige allgemeine Funktionen für die **Sphere2D**- und den **Sphere3DMapper** zusammen.

Arrow2DMapper Interpretiert die Daten der **VisionDomain** als Länge, Dicke, Richtung und/-oder Farbe von zweidimensionalen Pfeilen.

Arrow3DMapper Interpretiert die Daten der **VisionDomain** als Länge, Dicke, Richtung und/-oder Farbe von dreidimensionalen Pfeilen.

Axis_3D Stellt ein dreidimensionales Achsenkreuz dar.

DomainFineGridMapper Zeichnet die Kanten eines Feingitters.

DomainMapper Zeichnet die Kanten und Begrenzungsobjekte einer Domain.

I_Mapper Schreibt Methoden vor, die von Mapper-Implementierungen angeboten werden müssen.

NullMapper Erzeugt ein leeres Ergebnisbild.

Particle2DDefaultMapper Ein Standard-Mapper für 2D-Partikel-Verfolgung. Nur für Testzwecke.

ParticleTracing2DMapper Ein Mapper für 2D-Partikel-Verfolgung. Nur für Testzwecke.

ParticleTracingMapper Der Partikel-Verfolgungs-Mapper des Vision-Moduls. Simuliert die Bewegung von Partikeln im Strömungsfeld der **VisionDomain**.

ShadedPlotMapper Interpretiert die Werte der **VisionDomain** als Farbwerte innerhalb des Gitters.

Sphere2DMapper Interpretiert die Werte der **VisionDomain** als Durchmesser, Position und/-oder Farbe von Kreisen.

Sphere3DMapper Interpretiert die Werte der **VisionDomain** als Durchmesser, Position und/-oder Farbe von Kugeln.

C.7.11 devisor.vision.pipe.renderer

I_Renderer Dieses Interface schreibt Methoden vor, die von allen Renderer-Implementierungen angeboten werden müssen.

AbstractRenderer Eine Klasse, die allgemeine Funktionalitäten eines Renderers, wie das Initialisieren eines **Canvas3D**-Objekts anbietet, und als Grundlage für weitere Rendererimplementierungen dienen kann.

SimpleRenderer Der Standard-Renderer des VISIONModuls. Die vom Mapper erzeugte Szene wird mit zwei Lichtquellen versehen und so transformiert, daß sie vollständig sichtbar ist.

C.7.12 devisor.vision.stats

StatChart Diese Klasse stellt ein einzelnes Diagramm im **StatWindow** dar.

StatConfigurationDialog Hier wird ein Konfigurations-Dialog erzeugt.

StatManager Kapselt die komplette Funktionalität des Statistik-Moduls.

StatWindow Zeigt einen oder mehrere **StatChart**s an.

C.7.13 devisor.vision.util

EmptyParameterList Stellt eine leere Parameter-Liste zur Verfügung.

EmptyProblem Bereitet ein leeres Problem für die Modul-Kopnfiguration vor.

FileStreamer Erzeugt einen threadfähigen Stream aus einer Datei.

FindConfigurables Sucht die Parameterlisten aus allen in der **Configurables.xml**-Datei eingetragenen, das **I_Configurable**-Interface implementierenden Klassen zusammen.

LegendLabel Eine **JPanel**-Unterklasse, die eine für verschiedene Mapper benötigte Legende anzeigt.

LocaleHandler Ist für die Synchronisation der verwendeten Locales in allen Pipeline-Elementen zuständig.

Log4JConfigurator Initialisiert das VISION-interne Logging.

MouseConfig Datenstruktur zum Speichern der vom Benutzer getätigten Maus-Transformationen an der dargestellten Szene.

MouseManipulator Erlaubt das Drehen, Rotieren und Zoomen der im **VisionViewer** angezeigten Szene mit der Maus.

NoErrors Erzeugt eine leere Fehlerliste als Statusmeldung für CONTROL.

NetException, da sie logisch keinem anderen Paket zuzuordnen ist. Sie dient als Container für alle Exceptions, die innerhalb des Pakets auftreten können und an andere Module weitergegeben werden.

C.7.14 devisor.net.wrappers

Dieses Paket beinhaltet die zentralen Schnittstellen zwischen Modulen und dem Netzwerk.

ControlWrapper Diese Klasse stellt für jeden Protokollbefehl (s. Kap. D.3) eine Methode zur Verfügung, die eben diese Protokollsequenz durchführt. Außerdem wird eine Art *Posteingang* angeboten, in dem Huckepack-Nachrichten (siehe Kap. 3.2) abgelegt werden.

Anhang D

Format- und Protokollspezifikationen

D.1 Spezifikation des FEAST-Formates für die Gitterdefinition

Das existierende FEAST-Format wurde von der Projektgruppe auf den dreidimensionalen Fall erweitert und auch in 2D leichten Korrekturen und Ergänzungen unterzogen. Die Grundstruktur einer FEAST-Datei ist immer gleich:

1. Header mit Identifikatoren, Versionsnummern und Kommentaren,
2. Anzahlen der vorkommenden Knoten, Kanten, Elemente etc.,
3. Randbeschreibung,
4. Knotenliste,
5. Kantenliste,
6. In 3D: Facettenliste,
7. Elementliste.

Im FEAST-Format sind Leerzeilen nicht erlaubt, Kommentarzeilen dürfen beliebig eingefügt werden, sie beginnen dann mit dem Zeichen #. Das gesamte Format ist zeilenbasiert, stehen in einer Zeile mehrere Informationen (wie z.B. bei der Definition von Kanten, s.u.), so dient ein Leerzeichen als Trennung. Wichtig bei den Listen ist, daß jede Indizierung bei 1 beginnt: Bei n Elementen läuft die Nummerierung von 1 bis n und nicht 0 bis $n-1$. Wichtig: Die einzelnen hier beschriebenen Blöcke müssen immer durch Kommentarzeilen getrennt werden, ihr Inhalt ist dabei irrelevant.
Die hier spezifizierte FEAST-Version ist **Version 2.0**.

D.1.1 FEAST2D

Header Der Header enthält die Identifikationszeichenkette `FEAST`, den Modus `2D` und die Dateiversion.

```
FEAST          # ID-String
2D             # Modus
2              # Dateiversion major
0              # Dateiversion minor
```

Codebeispiel 83: Spezifikation des FEAST*-Formats in 2D: Header*

Anzahlen Der nächste Block enthält die Anzahlen an Parallelblöcken, Knoten, Elementen, Kanten und Rändern, jeweils in einer separaten Zeile.

```
4  # Anzahl Parallelbloecke
25 # Anzahl Macro-Knoten
16 # Anzahl Macro-Elemente
40 # Anzahl Macro-Kanten
2  # Anzahl Boundaries
```

Codebeispiel 84: Spezifikation des FEAST*-Formats in 2D: Anzahlen*

Randbeschreibungen So oft, wie im vorherigen Block angegeben, folgt nun eine Randbeschreibung. Sie besteht aus der Anzahl der (Teil-) Segmente, die den Rand definieren, gefolgt von einer Liste eben dieser Segmente. Als Segmenttypen stehen Linien und Kreise zur Verfügung, jeweils mit IDs 0 respektive 1. Eine Linie ist definiert durch ihren Start- und Endpunkt, jeweils in kartesischen Koordinaten, ein Kreis durch seinen Mittelpunkt, den Radius, und den Start- und Endwinkeln, gemessen im Bogenmaß gegen den Uhrzeigersinn von der ”3-Uhr-Position” (positive x-Achse) aus.

```
4  # Anzahl Segmente auf Boundary 1
0  # Typ des ersten Segments: Linie
0.0 0.0     # Startpunkt
1.0 0.0     # Endpunkt
0  # Typ des zweiten Segments: Linie
1.0 0.0     # Startpunkt
1.0 1.0     #Endpunkt
0  # Typ des dritten Segments: Linie
1.0 1.0     # Startpunkt
0.0 1.0     #Endpunkt
0  # Typ des vierten Segments: Linie
0.0 1.0     # Startpunkt
0.0 0.0     #Endpunkt
1  # Anzahl Segmente auf Boundary 2
1  # Typ des einzigen Segments: Kreis
1.0 1.0     # Mittelpunkt
1.0 0.0     # Radius + Platzhalter
0.0 3.1415  # Start- und Endwinkel
```

Codebeispiel 85: Spezifikation des FEAST*-Formats in 2D: Ränder*

Knoten Der nächste Block beinhaltet die Knotenliste. Hier ist zwischen inneren Knoten, die durch ihre kartesischen Koordidaten definiert werden, und Randknoten, die eindeutig durch den Index des Randes, auf dem sie liegen, und dem Parameterwert auf diesem Rand definiert werden, zu unterscheiden. Zur Erinnerung: Jedes Segment eines Randes entspricht einem Parameterintervall der Länge 1, das erste Segment ist über $[0, 1[$ parametrisiert, das zweite über $[1, 2[$ usw.

```
# Knotenliste
0.0  0.0 0  # X,Y,0 für inneren Knoten
            # dieser Knoten hat also Nummer 1
2.0 3.0 0   # innerer Knoten mit Nummer 2
0.25 0.0 1  # Parameterwert 0.25, Platzhalter 0, Randnummer 1
            # also Randknoten mit Nummer 3
0.5 0.0 1   # Randknoten mit Nummer 4
# usw
```

Codebeispiel 86: Spezifikation des FEAST*-Formats in 2D: Knoten*

Da alle Indices mit 1 beginnen, ist die Unterscheidung zwischen den Knotentypen eindeutig über die letzte Angabe (0 oder ≥ 1) zu treffen.

Kanten Kanten sind jeweils definiert über ihre Start- und Endknoten und haben zusätzlich noch einen Kantenstatus (je nachdem, ob sie innere oder äußere, d.h. Randkanten sind) und eine Randbedingung, *Dirichlet-Rand* oder *Neumann-Rand*.

```
# Kantenliste
1  2  1  0    # Index des Start- und Endknotens aus obiger
              # Knotenliste
              # Kantenstatus (0=none,1=inner,2=exter boundary)
              # Randbedingung (0=dirichlet,1=neumann)
2 7 0 0
# usw.
```

Codebeispiel 87: Spezifikation des FEAST*-Formats in 2D: Kanten*

Elemente Die Definition der Elemente beinhaltet deutlich mehr Informationen: Ein 2D-Element kann vom Typ Dreieck oder Viereck sein und ist definiert über die Knoten (gegen den Uhrzeigersinn), die es aufspannen, zusätzlich werden noch seine Kanten, die Elemente, die über die Kanten benachbart sind und die Elemente, die über die Knoten benachbart sind, angegeben. Es folgt (für die Durchführung der Berechnung auf Parallelrechnern) die Angabe des Parallelblocks und Matrixblocks, zu dem das Element gehört, bei Einprozessormaschinen sollte hier überall der Wert 1 eingetragen werden. Der letzte Abschnitt gibt die anisotrope Verfeinerung an: Dreieckselemente unterstützen in dieser Version keine Verfeinerungssteuerung, bei Viereckselementen werden in fünf Zahlwerten die folgenden Informationen codiert:

1. Die erste Zahl gibt das initiale Verfeinerungslevel, sowohl in x als auch in y-Richtung des Grobgitters an.

2. Die Verfeinerungssteuerung wird über 4 Bits gesetzt: Das niedrigwertigste Bit entscheidet, ob in x-Richtung verfeinert wird, das nächsthöhere, ob nach links oder rechts. Analog legen die beiden höherwertigen Bits fest, ob in y-Richtung verfeinert wird und ob nach unten oder oben. Gespeichert wird die Dezimalzahl, die dieser Bitsetzung entspricht.

3. die folgenden drei Zahlen geben die Verfeinerungsfaktoren an, die der zugrundeliegende Algorithmus benötigt.

Klar ist: Wird für den Verfeinerungsmodus 0, für den Level 0 und für die Faktoren jeweils 0.5 eingetragen, geht das anisotrope Element in eine handelsübliche Viereckszelle über. Ein Element wird also durch acht Zeilen im FEAST-Format spezifiziert.

```
# Makroliste
0        # Elementtyp: 0=Viereckselement, 1=Dreieck
1 2 7 6  # Indices der vier Knoten
1 2 3 4  # Indices der vier Kanten
0 2 5 0  # Indices der Kanten-Nachbar-Macros
0 6 0 0  # Indices der Knoten-Nachbar-Macros
1        # Index des Parallelblocks
1        # Index des Matrixblocks
1 0 0.5 0.5 0.5   # Refinementlevel   Refinementmodus
                  # Faktor1 Faktor2 Faktor3
# Es folgt Zelle Nummer 2
1        # Dreieck
2 3 8    # Indices der drei Knoten
5 6 7    # Indices der drei Kanten
0 3 6    # Indices der Kantennachbar-Zellen
0 7 5    # Indices der Knotennachbar-Zellen
1        # Parallelblock
2        # Matrixblock
1        # Verfeinerungslevel, keine Anisotropie
# usw.
```

Codebeispiel 88: Spezifikation des FEAST*-Formats in 2D: Elemente*

D.1.2 FEAST3D

Header Der Header unterscheidet sich nur durch die Angabe `3D` vom 2D-Format.

Anzahlen Zusätzlich wird zwischen der Anzahl der Kanten und der Elemente die Anzahl der Flächen angegeben:

```
4 # Anzahl Parallelbloecke
25 # Anzahl Macro-Knoten
16 # Anzahl Hacro-Zellen
40 # Anzahl Macro-Kanten
40 # Anzahl Macro-Flaechen
4  # Anzahl Boundaries
```

Codebeispiel 89: Spezifikation des FEAST*-Formats in 3D: Anzahlen*

Randbeschreibungen Jede Boundary besteht genau aus einem Rand-Grundelement, zur Verfügung stehen die vier folgenden Randtypen:

- Quader, definiert durch seine 8 Eckpunkte in der folgenden Reihenfolge: Die ersten vier Punkte definieren gegen den Uhrzeigersinn ein Viereck im Raum, die nächsten vier ebenfalls ein Viereck, in der Art, daß Knoten 1 und 5 durch eine Kante verbunden sind usw.
- Kugel, definiert durch den Mittelpunkt und den Radius.
- Zylinder, definiert durch den Mittelpunkt des Grundkreises, den Radius des Grundkreises und den Normalenvektor auf den Grundkreis entlang der Mittelachse des Zylinders. Die

Länge dieses Vektors definiert die Höhe des Zylinders. Dies entspricht einer Hesseschen Normalform für die Ebene, in der der Grundkreis liegt, und ist daher eindeutig.

- Dreiecksnetz, definiert durch die Angabe aller Eckpunkte und für jedes Dreieck die drei Punkte, die es aufspannen sowie der (maximal) drei Nachbardreiecke. Falls im Netz keine Nachbarschaftsinformationen zur Verfügung stehen, ist es möglich, jeweils die Zeilen, die sie beschreiben, wegzulassen. Ein separater Kontrollparameter steuert dieses Verhalten.

Die so beschriebenen Körper müssen alle konvex sein.
Da auch hier eine Parametrisierung analog zum 2D-Fall möglich sein soll, wird noch folgendes festgelegt: zu jedem Körper wird sein geometrischer Mittelpunkt mitgespeichert als Tripel von 3 Koordinaten. Dieser Schwerpunkt dient als Mittelpunkt einer Kugel, deren Radius so gewählt wird, daß sie vollständig im Inneren des Randobjektes liegt. O.B.d.A. sei der Mittelpunkt nun der Koordinatenursprung. Nun werden zwei Winkel α, β betrachtet: In der x, z-Ebene wird, bei Winkel $\alpha = 0$ parallel zur x-Achse, die Ebene gegen den Uhrzeigersinn durchlaufen, so daß jedem Winkel ein Punkt auf dem Kugeläquator entspricht. Analog wird in der y, z-Ebene bei Winkel $\beta = 0$ auf der y-Achse die Ebene gegen den Uhrzeigersinn (d.h. der Nullmeridian der Kugel) durchlaufen. Aus der geforderten Konvexität des Körpers folgt: Jedes Paar $(\alpha, \beta) \in [0, 2\pi[\times[0, 2\pi[$ ist (durch ”Strahlverlängerung”) eindeutig einem Punkt auf der Körperoberfläche zugeordnet und umgekehrt, diese Konstruktion definiert also eine eindeutige Parametrisierung. Aus historischen Gründen wird jedes Intervall nun noch auf den Bereich $[0, 1[$ skaliert.

```
0              # Typ 1  Quader
0.0 0.0 0.0    # Punkt 1: x1,y1,z1
0.0 0.0 0.0    # Punkt 2
0.0 0.0 0.0    # Punkt 3
0.0 0.0 0.0    # Punkt 4
0.0 0.0 0.0    # Punkt 5
0.0 0.0 0.0    # Punkt 6
0.0 0.0 0.0    # Punkt 7
0.0 0.0 0.0    # Punkt 8
1.0 1.0 1.0    # Mittelpunkt

1             # Typ 2  Kugel
0.0 0.0 0.0   # Mittelpunkt
0.0           # Radius
1.0 1.0 1.0   # Mittelpunkt

2             # Typ 3  Zylinder
0.0 0.0 0.0   # Mittelpunkt
0.0           # Radius
0.0 0.0 0.0   # Normalenvektor
1.0 1.0 1.0   # Mittelpunkt

3             # Typ 4  Dreiecksnetz
100           # Anzahl der Eckpunkte
0.0 0.0 0.0   # Eckpunkt 1
....
0.0 0.0 0.0   # Eckpunkt 100
100           # Anzahl der davon aufgespannten Dreiecke
0             # Nachbarschaftsinformationen:
              # 0=vorhanden, 1=nicht
1 2 3         # Indices der Knoten des ersten Dreiecks
              # Die Indizierung beginnt bei 1 und richtet sich
              # nach der Position der Knoten in obiger Liste
3 6 2         # Indices der Nachbardreiecke (falls oben 0)
              # eine 0 bedeutet: das Dreieck hat diesen
              # Nachbarn nicht
...
60 19 65      # Indices der Knoten des letzten Dreiecks
34 69 0       # Indices der Nachbardreiecke
              # dieses Dreieck hätte also nur zwei Nachbarn
1.0 1.0 1.0   # Mittelpunkt
```

Codebeispiel 90: Spezifikation des FEAST*-Formats in 3D: Ränder*

Knoten Die Definition der Knoten erfolgt analog zum zweidimensionalen Fall, nur daß jetzt, wie oben beschrieben, zwei Parameterwerte bzw. drei Koordinaten nötig sind.

```
# Knotenliste
0.0 0.0 0.0 0   # X,Y,Z,Platzhalter
0.0 3.1415 0 1  # alpha,beta,Platzhalter,Randnummer
```

Codebeispiel 91: Spezifikation des FEAST*-Formats in 3D: Knoten*

Die Unterscheidung erfolgt wieder über die letzte Zahl.

Kanten Kanten werden ebenfalls analog zum Zweidimensionalen über Start- und Endknoten, Kantenstatus und Randbedingung angegeben.

```
# Kantenliste
1 2 1 0      # Startknoten, Endknoten
             # Kantenstatus (0=none,1=inner,2=exter boundary)
             # Randbedingung (0=dirichlet,1=neumann)
```

Codebeispiel 92: Spezifikation des FEAST*-Formats in 3D: Kanten*

Flächen Eine Fläche ist definiert durch Angabe des Typs (Drei- oder Viereck), den drei oder vier Kanten, des Flächenstatus und der Randbedingung. Es stehen drei- oder viereckige Flächentypen zur Verfügung.

```
0 1 2 3 4 1 0 # Typ (0=quad)
              # Kante 1 - 4
              # Flächenstatus
              # (0=none,1=inner,2=exter boundary)
              # Randbedingung (0=dirichlet,1=neumann)
1 1 2 3 1 0   # Typ (1=tri)
              # Kante 1 - 3
              # Flächenstatus
              # (0=none,1=inner,2=exter boundary)
              # Randbedingung (0=dirichlet,1=neumann)
```

Codebeispiel 93: Spezifikation des FEAST*-Formats in 3D: Flächen*

Elemente Das Format unterstützt Hexaeder und Tetraeder-Elemente. Ein Hexaeder ist definiert durch seine 8 Knoten. Die Reihenfolge der Knoten und Kanten ist der folgenden Skizze zu entnehmen:
Der nächste Block gibt die Flächennachbarn an, danach die Indizes der noch nicht genannten Nachbarzellen, die durch nur eine Kante benachbart sind (ein negativer Wert bedeutet mehrere, -2 beispielsweise zwei Nachbarn), dies erzwingt für jede solche Kante einen Detailblock mit den Indizes eben dieser Kanten). Im letzten Schritt folgen die Knotennachbarn (Diagonalnachbarn) ebenfalls wieder mit dem Detailblock bei mehreren Nachbarn.

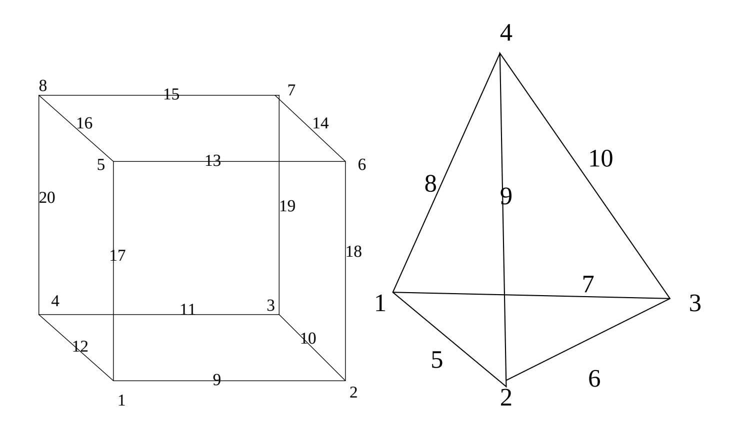


Abbildung D.1: Numerierung der Knoten und Kanten bei Hexa- und Tetraedern

Die Elementdefinition schließt mit der Angabe von Parallel- und Matrixblock und der anisotropen Verfeinerung. Sechs Bits sind zur Angabe der Verfeinerungssteuerung nötig, die beiden niedrigwertigen geben die Verfeinerung in x-Richtung an (links oder rechts), dann in y-Richtung (unten oder oben) und zum Abschluß in z-Richtung (vorne oder hinten). Es sind sechs Verfeinerungsfaktoren nötig.
Tetraeder werden vollkommen analog definiert, nur das es entsprechend weniger Knoten, Kanten und Nachbarn gibt. Auch eine anisotrope Verfeinerung ist analog zum zweidimensionalen Fall nicht möglich.

```
0                   # Typ (0=Hex, 1=Tetra)
1 2 7 6 9 10 11 12 # Knoten 1 bis 8
1 2 3 4 5 6 7 8 9 10 11 12 # Kanten 1 bis 12,
1 2 3 4 5 6         # Seitenflächen 1-6,
0 2 5 0 0 0         # Indizes der 6 durch eine Fläche
                    # benachbarter Zellen
0 2 5 0 0 0 -3 0 -2 0 0 0  # Indizes der noch nicht
                    # genannten Nachbarzellen, die
                    # durch nur eine Kante benachbart sind
1 2 3               # Nachbarn fuer Edge 7 (Position der Kante
                    # in obiger Liste mit negativer Bewertung)
1 2                 # Nachbarn fuer Edge 9
0 6 0 0 -4 0 0 0    # Indizes der noch nicht genannten
                    # Diagonalnachbarn
1 2 3 4             # Diagonalnachbarn fuer Punkt 5
1                   # Index des Parallelblocks des Makros
1                   # Index des Matrixblocks des Makros
1 0 0.5 0.5 0.5 0.5 0.5 0.5
                    # Refinementlevel   Refinementmodus
                    # Faktor1 .... Faktor6
# nächstes Element
1                   # Typ: Tetra
1 2 3 4             # Knoten
6 7 8 9 4 3         # Kanten
3 4 5 6             # Seitendreiecke
55 66 77 88         # Flächennachbarn
2 3 4 5 -2 7        # Kantennachbarn
44 33               # Details für Kante mit Index 5
2 3 -2 -3           # Diagonalnachbarn
2 7                 # Details für Kante mit Index 3
3 4 5               # Details für Kante mit Index 4
2                   # Parallelblock
3                   # Matrixblock
5                   # Level
```

Codebeispiel 94: Spezifikation des FEAST*-Formats in 3D: Elemente*

D.2 Spezifikation des GMV-Formats

Die folgende Spezifikation ist [26] entnommen. Auf eine Übersetzung wurde verzichtet.

D.2.1 Input Specifications

The format for GMV's input file follows. Please note that there are relatively few required entries, most data is optional, and keywords are used to identify its type. The data on the file can be either formatted ASCII or IEEE unformatted (but not both). Keywords are italicized and data names are in boldface. Example names for variables or flags are in double quotation marks. A description of the input line follows the data names or keywords. Only gmvinput, the file type, node data, cell or face data, and endgmv are required, everything else is optional,

however, except for material and velocity keywords, each keyword may be used only once. For example, a second variable list is not allowed. For IEEE unformatted files, keywords must be written as eight character words.

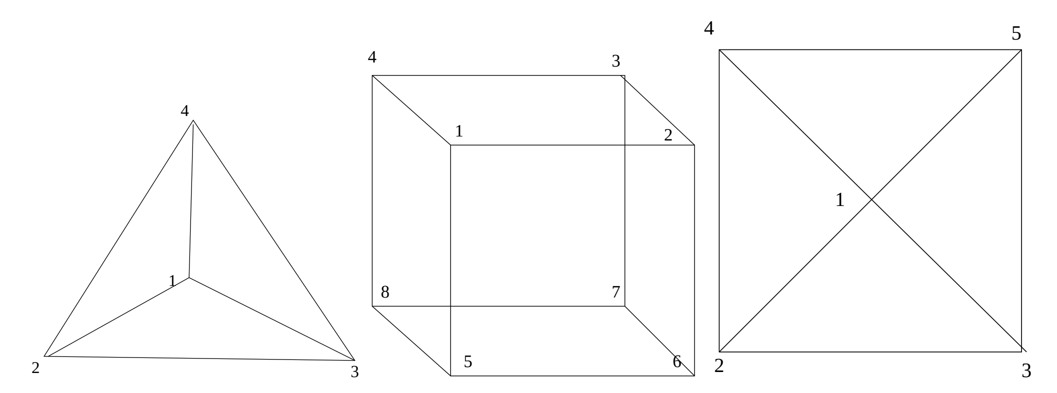


Abbildung D.2: cell numbering (1): tet, hex and pyramid

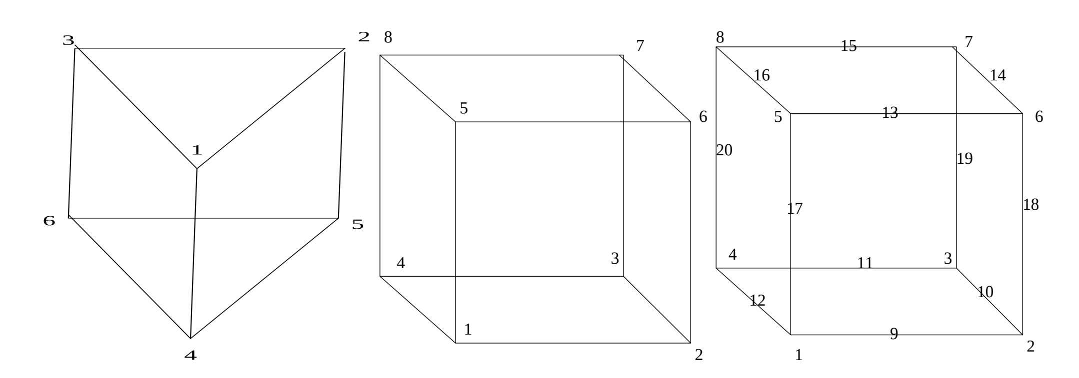


Abbildung D.3: cell numbering (1): prism, phex8 and phex20

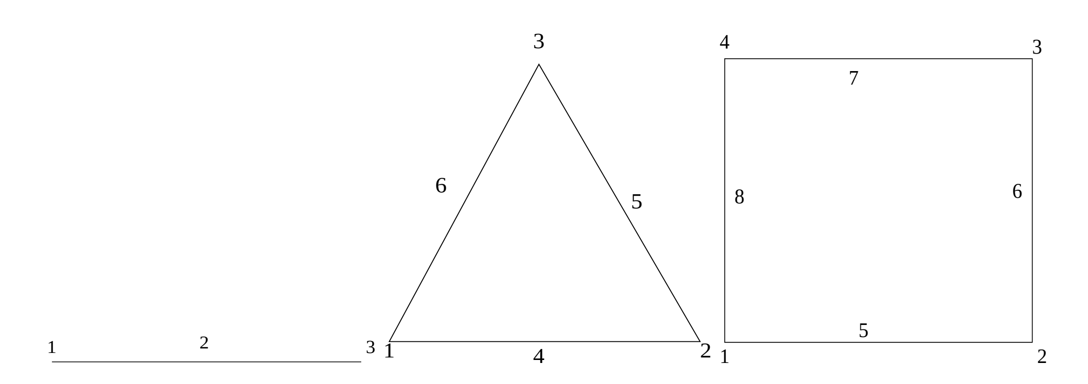


Abbildung D.4: cell numbering (1): 3line, 6tri, 8quad

gmvinput **file_type** The first line identifies the file as a GMV input file with the file_type being either ”ascii”, ”ieee”, ”ieeei4r4”, ”ieeei4r8”, ”iecxi4r4”, or ”iecxi4r8”.

nodes **nnodes** Node points and number of points.
x(nnodes) Float, x coordinates.

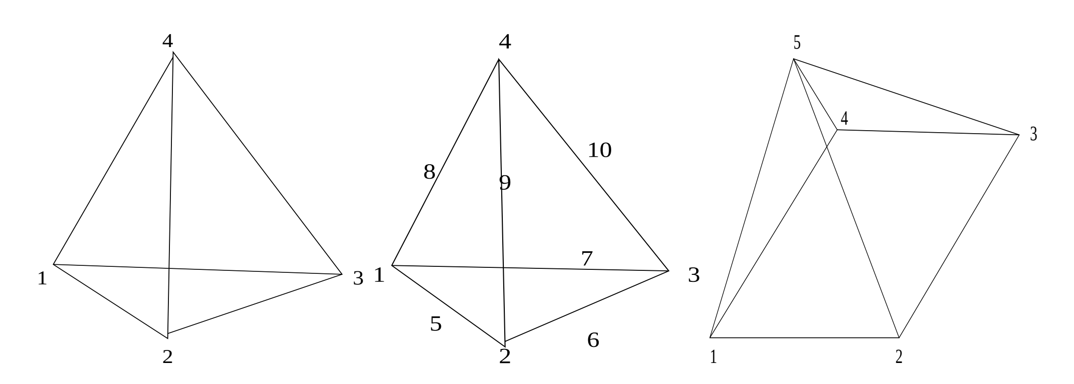


Abbildung D.5: cell numbering (1): ptet4, ptet10 and ppyrmd5

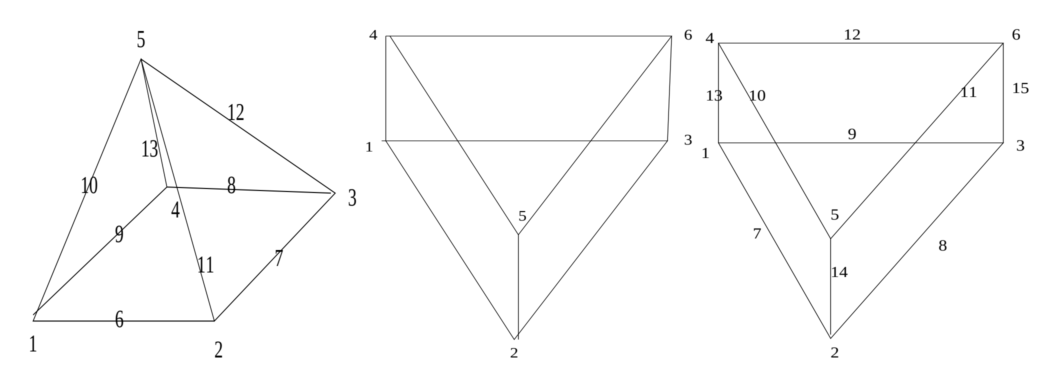


Abbildung D.6: cell numbering (1): ppyrmd13, pprism6 and pprism15

y(nnodes) Float, y coordinates.
z(nnodes) Float, z coordinates.

OR (for a structured regular brick mesh)

nodes-1 **nxv nyv nzv** Dimensions of structured regular brick mesh.
x(nxv) Float, x coordinates.
y(nyv) Float, y coordinates.
z(nzv) Float, z coordinates.

OR (for a logically rectangular brick mesh)

nodes-2 **nxv nyv nzv** x(nxv * nyv * nzv) Float, x coordinates.
y(nxv * nyv * nzv) Float, y coordinates.
z(nxv * nyv * nzv) Float, z coordinates.

nodev **nnodes** Node points and number of points. (Instead of nodes.)
x(1) y(1) z(1) Float, first x, y, and z coordinates.
x(2) y(2) z(2) Float, second x, y and zcoordinates.
. . .
x(nnodes) y(nnodes) z(nnodes) Float, last x, y, and z coordinates.

***cells* ncells** Cell data and number of cells.

***cell_type* number of elements** Cell data; format depends on cell type, vertex or face data. See examples. Allowable cell types (Note: cell types can be mixed except for vface3d and vface2d):

***general* nfaces** General type, number of faces in cell.
nverts(nfaces) Number of vertices per face.
vertex_ids(sum(nverts)) Integer list of node numbers that define the polygonal faces
Note: the general type can be used to define any cell volume.

***line* 2** Linear cell, with 2 vertex ids.
verts(2) vertex ids.

***tri* 3** Triangular cell, with 3 vertex ids.
verts(3) vertex ids.

***quad* 4** Quadrilateral cell, with 4 vertex ids.
verts(4) vertex ids.

***tet* 4** Tetrahedral cell, with 4 vertex ids.
verts(4) vertex ids.

***hex* 8** hexahedral cell, with 8 vertex ids.
verts(8) vertex ids.

***phex8* 8** hexahedral cell, with 8 vertex ids, Patran order.
verts(8) vertex ids.

***phex20* 20** hexahedral cell, with 20 vertex ids, Patran order.
verts(20) vertex ids.

***prism* 6** Prism cell, with 6 vertex ids.
verts(6) vertex ids.

***pyramid* 5** Pyramid cell with 5 vertex ids.
verts(5) vertex ids. Note: the ordering of the vertices for tet, hex, prism and pyramid is shown in Fig. 9 1. The tri and quad cells are two dimensional entities that employ a sequential vertex numbering scheme around the periphery of the cell.

***vface3d* nfaces** Vface3d type, number of faces in cell.
vfaces(nfaces) vface ids in vface keyword (see below). Note: vface3d type cannot be mixed with other cell types.

***vface2d* nfaces** Vface2d type, number of faces in cell.
vfaces(nfaces) vface ids in vface keyword (see below). Note: vface2d type cannot be mixed with other cell types. Vface2d faces are 2D faces, i.e. edges.

***vfaces* nfaces** Vface data, number of faces. Followed by nfaces lines of:
nverts Number of vertices per face.
pe_no Processor number of the face (1 if single processor).
oppface Opposite face number.
opp_pe_no Processor number of the opposite face (1 if single processor).
cellno Cell number of the cell the face is part of.
vertex_ids Integer list of node numbers that define the polygonal faces.

***faces* nfaces ncells** Face data, number of faces, and number of cells. Followed by nfaces lines of:
nverts Number of vertices per face.
vertex_ids Integer list of node numbers that define the polygonal faces.
cellno1 Cell number of the cell to the left of the face.
cellno2 Cell number of the cell to the right of the face The cell numbers must be between 0 and ncells where 0 indicates no cell exists on that side of the face.
Note, face data includes cell information; do NOT mix both cells and faces within one GMV input file.

nodeids Alternate node id numbers for display
ids(nnodes) Integer, alternate node ids.

cellids Alternate cell id numbers for display
ids(ncells) Integer, alternate cell ids.

faceids Alternate face id numbers for display
ids(nfaces) Integer, alternate face ids.

***material* nmats data_type** Material data, number of materials, and data type (0=cells, 1=nodes).
matnames(nmats) 8 character material names.
matids(ncells) Integer, material ids for cells.
or
matids(nnodes) Integer, material ids for nodes.

***velocity* data_type** Velocity data, and data type (0=cells, 1=nodes, 2=faces)
u(ncells) |
v(ncells) | For cells.
w(ncells) |
or
u(nnodes) |
v(nnodes) | For nodes.
w(nnodes) |
or
u(nfaces) |
v(nfaces) | For faces.
w(nfaces) |

variable Keyword indicating that other cell or node data sets follow. The data sets have the form:
"anyname" data_type An eight character name for the data, the data type (0=cells, 1=nodes, 2=faces)
data(ncells, nnodes or nfaces) array of float data.
Examples (DO NOT use quotes in actual file):

"density" 0
density_data(ncells)

”temp” 0
temp_data(ncells)

”pressure” 1
pressure_data(nnodes)

”flow” 2
flow_data(nfaces)

endvars Keyword indicating end of variable data.

subvars Keyword indicating that other cell or node data sets for a subset of the mesh follows.
The data sets have the form:
”anyname” data_type num_elem An eight character name for the data, the data type (0=cells, 1=nodes, 2=faces), the number of elements in the subset.
elem_ids(num_elem) array of integer cells, nodes or faces in the subset.
data(num_elem) array of float data.
Examples (DO NOT use quotes in actual file):

”bdryt” 0 10
5 6 9 13 28 101 150 181 210 300
0.5 0.5 1.2 3.4 6.1 0.5 1.8 0.1 -2.2 -3.3

”intp” 1 5
1 2 3 6 7
9.8 10.0 11.0 0.2 2.2

”facefl” 2 8
20 30 40 50 60 70 90 100
100.0 200.0 300.0 10.0 12.2 8.5 2.8 0.0

endsubv Keyword indicating end of subvars data.

flags Keyword indicating that selection flag data sets follow. The data sets have the form:
”anyname” ntypes data_type Flag name, number of flag types, and data type (0=cells, 1=nodes).
flagnames(ntypes) 8 character flag type names.
iflag(ncells) Integer, flag ids for cells.
or
iflag(nnodes) Integer, flag ids for nodes.
Examples:

”nodetype” 4 1
”inactive” ”interior” ”interfac” ”boundary”

node_data(nnodes)

”cnstrain” 3 0
”static” ”piston” ”air”
cnst_data(ncells)

endflag Keyword indicating end of flag data.

polygons Keyword indicating surface polygon data follows. (eg. interface or boundary faces.)
material_no nverts x(nverts) y(nverts) z(nverts)
Where:
material_no Integer number related to material data.
nverts No. of vertices.
x(nverts) x coordinate of polygon vertices.
y(nverts) y coordinate of polygon vertices.
z(nverts) z coordinate of polygon vertices.
This data is repeated for all polygons.

endpoly Keyword to indicate end of polygon data.

tracers **ntracers** Tracer points and the number of tracers input.
x(ntracers) Float, x coordinates.
y(ntracers) Float, y coordinates.
z(ntracers) Float, z coordinates.
Followed by trace data of the form:
”anyname” An eight character name for the data.
data(ntracers) array of float data.
Examples:

”temp”
temp_data(ntracers)

”pressure”
pressure_data(ntracers)

endtrace Keyword indicating end of variable data.

traceids Alternate tracer id numbers for display
ids(ntracers) Integer, alternate tracer ids.

probtime **ptime** Keyword and floating point problem time value.

cycleno **cycleno** Keyword and integer problem cycle number.

surface **nsurfaces** Surface facet data and number of facets. Followed by nsurfaces lines of:
nverts Number of vertices per facet.
vertex_ids Integer list of node numbers that define the polygonal facet.

surfmats Surface material data.
matids(nsurfaces) Integer, material ids for surface facets. Note, surface data must be input before surfmats.

surfvel Surface velocity data.
u(nsurfaces)
v(nsurfaces)
w(nsurfaces)

surfvars Keyword indicating that other surface field data sets follow. The data sets have the form:
”anyname” An eight character name for the data.
data(nsurfaces) array of float data.
Examples (DO NOT use quotes in actual file):

”density”
density_data(nsurfaces)

”temp”
temp_data(nsurfaces)

endsvars Keyword indicating end of variable data

surfflag Keyword indicating that surface selection flag data sets follow. The data sets have the form:
”anyname” ntypes Flag name and number of flag types.
flagnames(ntypes) 8 character flag type names.
iflag(nsurfaces) Integer, flag ids for surfaces. Note, surface data must be input before surfmats.
Examples:

”surftype” 4
”inactive” ”interior” ”interfac” ”boundary”
sflg_data(nsurfaces)

”cnstrain” 3
”static” ”piston” ”air”
cnst_data(nsurfaces)

endsflag Keyword indicating end of flag data.

surfids Alternate surface id numbers for display ids(nsurfaces) Integer, alternate surface ids.

groups Keyword indicating that user defined group sets follow. The data sets have the form:
”anyname” An eight character name for the group,
data_type The Data type (0=cells, 1=nodes, 2=faces, 3=surfaces),
nelem The number of elements in the group.

data(nelem) array of integer cells, nodes or faces, depending on type.
Examples (DO NOT use quotes in actual file):

”cells1” 0 5 1 4 7 10 12
”nodes1” 1 8 5 7 10 30 45 50 55 62
”faces1” 2 4 1 4 12 18

endgrp Keyword indicating end of group data

comments Keyword indicating that ASCII comments follow.

endcomm Keyword indicating end of comments.

***codename* ”anyname”** The name of the code that generated the file where ”anyname” is the eight character name of the code.

***codever* ”version”** The version of the code that generated the file where ”version” is the eight character version of the code.

***simdate* ”date”** The date the file was generated where ”date” is in the form $mm/dd/yy$.

endgmv Keyword signifying the end of the input file.

D.2.2 Input Data Details

Header The header line contains the ***gmvinput*** keyword and the character variable file_type which contains either ”ascii”, ”ieee”, ”ieeei4r4”, ”ieeei4r8”, ”iecxi4r4” or ”iecxi4r8”. The ASCII file type indicates that the file was written as a formatted ASCII file; the file will be read using list directed I/O, so there must be at least one space between data elements. The ieee, ieeei4r4 and ieecxi4r4 file types indicate the file was written as an unformatted file with IEEE single precision floating point, and 32 bit integers. ieeei4r8 and iecxi4r8 indicate that all floating point data is 64 bit. The iecxi4r4 and iecxi4r8 file types indicate that character data is 32 characters long while the other ieee types have eight characer data. Keywords are still eight characters long.

Nodes The ***nodes*** keyword describes the beginning of cell node data points and the variable ***nnodes*** on this line are the number of nodes (i.e. the length of the node data arrays that follow). The next three lines are the three floating point arrays that represent the X, Y, and Z coordinates of the nodes. The ***nodes*** keyword has three alternate forms. The first is used to generate a structured, regular brick mesh. Entering **-1** for the number of nodes signifies this alternate syntax. After **-1** on the same line are the dimensions of the mesh; first the number of nodes along the X axis, then the number along Y, and the number along the Z axis. The three lines that follow contain the X, Y, and Z coordinates of the nodes along each axis, which will be used by GMV to generate the entire mesh. Note: because GMV uses this information to generate a large mesh of cells, the number of cells specified with the ***cells*** keyword must be zero.
The second alternate syntax for the ***nodes*** keyword is used to generate a logically rectangular structured mesh. Entering **-2** for the number on nodes signifies this alternate syntax. After **-2** and on the same line are the dimensions of the mesh; first the number of nodes along the X

axis, then the number along Y, and the number along the Z axis. The three lines that follow contain the X,Y, and Z coordinates of the nodes for all nodes (nx*ny*nz), which will be used by GMV to generate the entire mesh. Note: because GMV uses this information to generate a large mesh of cells, the number of cells specified with the ***cells*** keyword must be zero. For any meshes that are closed, you need to repeat the necessary nodes for closure. Node data input may also be performed by consulting a remote file (see the ***fromfile*** description below).

Nodev The ***nodev*** keyword is an alternate form of the "nodes" keyword. Use nodev to input the node coordinates as a triplet on each line. There will be nnodes lines of the x, y and z coordinate on each line. The nodev keyword cannot process a structured brick mesh.

Cells The ***cells*** keyword indicates the beginning of cell descriptions. The variable ncells on this line are the number of cell descriptors that follow. There are six standard cell types that GMV can read, ***line, tri, quad, tet, hex, pyramid, prism, phex8, phex20, 3line, 6tri, 8quad, ptet4, ptet10, ppyrmd5, ppyrmd13, pprism6***, and ***pprism15***. The type is followed by the number of vertices contained in the cell, 2 for line, 3 for tris, 4 for quads, 4 for tets, 8 for hex, 5 for pyramid, 6 for prisms, 8 for phex8, 20 for phex20, etc. The next line contains the node numbers for the cell vertices. The vertex ordering for selected standard cell types is shown in Fig. 9 1. This ordering must be followed in order to ensure outward pointing normals. The tri and quad cells are simple two dimensional entities that employ a sequential vertex numbering scheme counterclockwise around the periphery of the cell.
The ***general*** cell type is available for nonstandard cells. These cells are described by their faces. The nfaces variable indicates the number of faces for the cell. The next line of data is the number of vertices for each cell face. The third line of the set contains the node numbers of the vertices for each face for all faces. The integer array size for the nodes will be the sum of the vertices for the cell faces. The faces do not have to be specified in any order. However, the vertices for each face must be specified in an order that describes the face polygon and generates an outward normal using the right hand rule.
The ***vface3d*** cell type is available for nonstandard 3D cells. These cells are described by their face numbers within the vface keyword. The nfaces variable indicates the number of faces for the cell. If ***vface3d*** is used, all cell types must be ***vface3d***.
The ***vface2d*** cell type is available for nonstandard 2D cells. These cells are described by their face numbers within the vface keyword. The nfaces variable indicates the number of faces for the cell. If ***vface2d*** is used, all cell types must be ***vface2d***. The faces in ***vface2d*** are 2d faces (edges).
The node and cell data are required and must be in order, although the number of cells can be zero if no cells exist. Note, there is no external numbering for the nodes and cells; the order of input is the numbering sequence for both nodes and cells.
Cell data input may also be performed by consulting a remote file (see the ***fromfile*** description below).

Vfaces The ***vfaces*** keyword indicates the beginning of polygonal face descriptions related to the ***vface3d*** or ***vface2d*** cell types. The variable ***nfaces*** on this line is the number of face descriptors that follow. There are nfaces descriptor lines that contain the number of vertices in the face, the processor number for the face, the opposite face number, the processor number

for the opposite face, the cell in which this face is part of, and the list of node numbers of the vertices for the face. Order the face vertices to generate an outward normal using the right hand rule. If the face describes a ***vface2d*** face, then it must contain only two vertices.

Vface data input may also be performed by consulting a remote file (see the ***fromfile*** description below).

Faces The ***faces*** keyword indicates the beginning of polygonal face descriptions and their associated cells. The variable ***nfaces*** on this line is the number of face descriptors that follow, and the variable ***ncells*** is the number of cells in the problem There are nfaces descriptor lines that contain the number of vertices in the face, the nodes that define the face, the cell that is to the left of the face, and the cell to the right of the face. Each polygonal face can be a part of up to two cells,if the face is part of only one cell, a cell number of 0 (zero) must be input.

Face data input may also be performed by consulting a remote file (see the ***fromfile*** description below).

The ***cells*** keyword and the ***faces*** keyword must NOT both exist in the same GMV input file since they are alternate forms of the same mesh information.

Nodeids The keyword ***nodeids*** indicates that an optional list of alternate node id numbers follows. These alternate id numbers are used for display and reference purposes within GMV. Enter nnode integers.

Cellids The keyword ***cellids*** indicates that an optional list of alternate cell id numbers follows. These alternate id numbers are used for display and reference purposes within GMV. Enter ncell integers.

Faceids The keyword ***faceids*** indicates that an optional list of alternate face id numbers follows. These alternate id numbers are used for display and reference purposes within GMV. Enter nface integers.

Materials The keyword ***material***, denoting material data, is an optional but highly recommended input data type. Up to 1000 materials are allowed. On the keyword line are the variables nmats (1 to 1000) and data_type (0 means the material data is cell centered and 1 means the material data is node centered). The next line of data is the eight character names given to the nmats materials. Finally, the last line of material data is the cell or node centered material ids; this is an integer array. Material data is necessary if surface polygons exist.

Up to two material keywords can be entered, one for cells and one for nodes. However, both keywords must contain the same material names in the same order.

Material data can be used to distinguish between different classes of cell or node data besides the normal engineering material definitions. For example, the material data can be density layers for an ocean model, horizons in seismic data, or rock layers in a reservoir model.

Material data input may also be performed by consulting a remote file (see the ***fromfile*** description below), but only if the remote file contains only one ***material*** keyword.

Velocities The keyword ***velocity*** indicates that optional velocity data follows. Again the ***data_type*** value of 0 indicates cell centered velocities, a value of 1 indicates node centered velocities and a value of 2 indicates face centered velocities. The next three lines of data are the u (x component), v (y component), and w (z component) velocity floating point arrays. Cell centered velocities will be averaged and saved as node centered velocities. Also, speed and kinetic energy variable fields will be automatically generated and added to the end of the input variable fields. Face centered velocities can only be entered when faces or vfaces are used to define cells.
Up to three velocity keywords can be entered, one for cells, one for nodes, and one for faces.

Variable data fields The ***variable*** keyword is used to denote the beginning of any other cell, node or face data fields. The data are entered as a group for each field variable. Up to 250 different field variables are allowed, and each field variable is named by the user. The ***endvars*** keyword is used to end the field data input. Each field data variable is defined by two input lines. The first line contains the eight character name of the variable and the data_type of the field (0 cell data, 1 node data, 2 face data). The second line is the floating point array for the cell or node data. Cell centered field data will be averaged and stored as node centered data. Face centered data can only be entered when faces or vfaces are used to define cells.

Subset variable data fields The ***subvars*** keyword is used to denote the beginning of cell, node or face data fields that are defined for a subset of the mesh. The data are entered as a group for each field variable for the defined elements. These fields are added to the variable field list. The ***endsubv*** keyword is used to end the field data input. Each field data variable is defined by three input lines. The first line contains the eight character name of the variable, the data_type of the field (0 cell data, 1 node data, 2 face data) and the number of elements to define and set data for. The second line is the list of elements (nodes, cells or faces) that will carry the field data. The third line is the floating point array for the field data. Elements not defined in the list will carry a value that is less than the minimum value entered. Face centered data can only be entered when faces or vfaces are used to define cells.

Selection Flags The ***flags*** keyword means that integer selection flag data sets follow. Up to 10 different types of selection flags and up to 1000 different flag values per flag are allowed. These data sets can be any type of integer data that can be used to select a node or cell for display purposes. The names for the flags and for the flag types are placed in selection buttons in a menu. The integer data must be a number between 1 and ntypes (1 to 1000). The ***endflag*** string ends the flag data.
Flag data input may also be performed by consulting a remote file (see the ***fromfile*** description below).

Polygons The ***polygons*** keyword indicates that surface polygons data follows. The surface polygons can be interface or boundary polygons for a material. Each line describes one polygon. The line contains the material number (1 to ***nmats***) associated with the polygon, the number of vertices in the polygon and the x, y, z arrays that define the vertices for the polygon. The ***endpoly*** string terminates the polygon data.

The ***polygons*** keyword can be used to describe any surface a simulation can generate. Be sure to give each surface a material number and that this material number has a material name listed under the ***materials*** keyword.
Polygon data input may also be performed by consulting a remote file (see the ***fromfile*** description below).

Tracers The ***tracers*** keyword indicates that tracer particle data (or any point data other than node data) follow. The ***ntracers*** variable following the keyword is the number of tracers that are input. The next three lines are the x, y, and z floating point coordinates of the tracers. Following the coordinates are the variable data fields for the tracers. The data is entered as a group for each field variable. Up to 40 different field variables are allowed. Each tracer field data variable contains an eight character variable name followed by a floating point data array. The ***endtrace*** string terminates the tracer data.

Traceids The keyword ***traceids*** indicates that an optional list of alternate tracer id numbers follows. These alternate id numbers are used for display and reference purposes within GMV. Enter ***ntracer*** integers.

Problem Time The ***probtime*** keyword is followed by a floating point number that represents the simulation problem time. This value is displayed at the top right corner of the main viewer.

Cycle Number The ***cycleno*** keyword is followed by an integer number that represents the familiar cycle number. This value is displayed at the top left corner of the main viewer.

Surface The ***surface*** keyword indicates the beginning of polygonal facet descriptions of surfaces. The variable ***nsurface*** on this line is the number of facet descriptors that follow. There are nsurface descriptor lines that contain the number of vertices in the facet and the nodes that define the facet.

Surface materials The keyword ***surfmats***, denoting surface material data, is an optional but highly recommended input data type. Up to 128 materials are allowed. After the keyword line of surface material data is the surface facet centered material ids; this is an integer array.

Surface velocities The keyword ***surfvel*** indicates that optional surface velocity data follows. The next three lines of data are the u (x component), v (y component), and w (z component) surface velocity floating point arrays. Speed and kinetic energy variable fields will be automatically generated and added to the end of the surface input variable fields.

Surface variable data fields The ***surfvars keyword*** is used to denote the beginning of any other surface data fields. The data are entered as a group for each field variable. Up to 100 different field variables are allowed, and each field variable is named by the user. The ***endsvars*** keyword is used to end the field data input. Each field data variable is defined by two input lines. The first line contains the eight character name of the variable. The second line is the floating point array for the surface data.

Surface Selection Flags The ***surfflag*** keyword means that integer selection flag data sets follow. Up to 10 different types of selection flags and up to 128 different flag values per flag are allowed. These data sets can be any type of integer data that can be used to select a node or cell for display purposes. The names for the flags and for the flag types are placed in selection buttons in a menu. The integer data must be a number between 1 and ntypes (1 to 128). The ***endsflag*** string ends the surface flag data.

Surfids The keyword ***surfids*** indicates that an optional list of alternate surface id numbers follows. These alternate id numbers are used for display and reference purposes within GMV. Enter ***nsurface*** integers.

Element groups The ***groups*** keyword is used to denote the beginning sets of arbitrary elements. A group is a named set of nodes, cells, faces, or surfaces. Up to 1000 different groups per type are allowed, and each group is named by the user. The ***endgrp keyword*** is used to end group input. Each group is defined by an eight character name, the element type (node, cell, face, or surface), the number of elements in the group and the list of element numbers.

Comments The ***comments*** keyword means that ascii comments follow. The ***endcomm*** string ends the comments. Be sure that there is a blank before and after endcomm.

Codename, codever, simdate The ***codename, codever, and simdate*** keywords allows the name and version of the code, as well as the date the simulation generated the file to be added to the file for identification purposes. These keywords can be placed before the ***nodes*** keyword.

D.2.3 Reading some GMV data from a different file

Certain GMV keywords and their data can be read in from a different GMV input file. These keywords are ***nodes, cells, faces, material, flags, polygons, nodeids, cellids*** and ***faceids***. The format for all the keywords is similar to the following:
***nodes fromfile* "filename"**
This syntax is used within the scope of the current GMV file to instruct GMV that keyword data is located in a ***fromfile*** specified by ***"filename"*** . Filename is a user supplied character string that must be enclosed by double quotes. The use of this keyword form specifies that the fromfile will contain the pertinent data in the same format and context as would be used in the main GMV file; the fromfile must be a valid GMV format file. When GMV encounters this keyword form, main file processing stops, and the fromfile is opened and searched for the applicable data. Once the data is input, the fromfile is closed, and main GMV file processing continues.
Fromfiles are useful within GMV for displaying animation sequences and the production of movies. In many animation sequences, much of the data remains unchanged between frames (for example, nodes, cells, faces, material, flags, and polygon data may remain constant if the problem domain and physical geometry of the problem does not change between frames). Constructing distinct and complete GMV files of each frame consumes much disk space needlessly; the fromfile capability allows the placement of constant data (i.e., one or more of node, cell, face, material, flag, and polygon data) within a single file that will be repeatedly be referenced by several GMV input files. This constant file is tagged a fromfile in this implementation.

D.2.4 Sample input data

The following is a sample GMV input file in ASCII format. It includes most of the features and commands GMV allows. When read in, the file creates a cube with several other different cell types attached to it. The additional cells are: one tetrahedral cell, one prism cell, one pyramid cell, and one general cell. The general cell has ten faces, and could be described as an octagonal prism. There is a large cube constructed from square polygons, each with a different material, that encloses all of the cells. In addition, variables, tracers, and flags with arbitrary data have been included so that you may see the format for entering such elements into an input file. The keywords are italicized. In addition, blank lines have been inserted between major elements of the file for clarity, but these are not necessary in a real input file. The input data follows:

```
gmvinput ascii
comments
A set of comments
endcomm
nodes 28
0 50 50 0 0 0 50 50
25 25 25 25 50 50 50 50 50 50
  50 50 80 80 80 80 80 80 80 80
0 0 0 0 50 50 50 50 75 0 50  25 16.7 33.4 50 50 33.4
   16.7 0 0 16.7 33.4 50 50 33.4 16.7 0 0 50 50 0
0 50 0 0
50 25 80 80 65 0 0 16.7 33.4 50 50 33.4
   16.7 0 0 16.7 33.4 50 50 33.4
16.7
cells 5
hex 8
1 2 3 4 5 8 7 6
pyramid 5
9 5 6 7 8
prism 6
10 1 2 11 5 8
tet 4
12 1 2 10
general 10
8 8 4 4 4 4 4 4 4 4
13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
19 27 28 20 20 28 21 13 13 21 22 14 14 22 23 15
23 24 16 15 24 25 17 16 25 17 18 26 26 18 19 27
material 6 0
mat1
mat2
mat3
mat4
mat5
mat6
1 2 3 4 5
```

Codebeispiel 95: GMV *sample input (1)*

```
polygons
1 4  -100 100 100 -100 100 100 -100 -100 100 100 100 100
2 4  -100 -100 100 100 -100 100 100 -100- 100 -100 -100 -100
3 4  -100 100 100 100 -100 100 100 100 100 100 -100 -100
4 4  -100 100 100 -100 -100 -100 -100 -100 100 100 -100 -100
5 4 100 100 100 100 -100 100 100 -100 100 100 -100 -100
6 4 100 -100 -100 -100 -100 -100 100 100 100 -100 -100 100
endpoly
tracers 10
0 20 40 60 80 100 120 140 160 180
0 20 40 60 80 100 120 140 160 180
0 20 40 60 80 100 120 140 160 180
pressure
0 5 10 15 20 25 30 35 40 45
temp
45 40 35 30 25 20 15 10 5 0
density
0 5 10 15 20 25 30 35 40 45
endtrace
traceids
2 4 10 12 15 18 20 21 22 23
velocity 0
0 0 5 0 0
0 5 5 0 10
5 5 5 10 10
variable
pressure 0
0 5 10 15 20
temp 0
0 5 10 15 20
density 0
0 5 10 15 20
endvars
flags
flagtype 3 0
good bad ugly
1 2 3 2 1
stufftype 3 0
bing bang boom
1 2 3 2 1
```

Codebeispiel 96: GMV *sample input (2)*

D.3 Spezifikation des Netzwerkprotokolls

D.3.1 Der Aufbau des Protokolls

Das Netzwerkprotokoll spezifiziert das, was konkret über das Netz verschickt wird. Es sind, wie oben spezifiziert ist, Zeichenketten, die hierarchisch aufgebaut sind.

Das Protokoll ist stets gleichartig aufgebaut. Zuerst kommt der `Header` (Codebeispiel 97). Danach folgt der eigentliche Protokollblock, der entweder einen Protokoll- oder einen Datenblock (Kap. 97) enthält. Wenn allerdings Fehler an einem beliebigen Teil der Kommunikation auftreten, dann besteht der Protokollblock aus einer `ListOfErrors` (Codebeispiel 99), die die aufgetretenen Fehler auflistet. Diese Liste wird aber auch benutzt, um Statusmeldungen innerhalb eines Kommandos zu übertragen.
Zum Schluß kommt noch der `Footer` (Codebeispiel 98), der das Ende der Kommunikationssequenz einleitet und einige Statusmeldungen enthält.

Die Kommandos treten stets paarweise auf, ein Kommando und deren Anwort (Kap. 3.2.1). Sie können eine `ParameterListe` enthalten, in der die `Parameter` (Kap. D.3.2) des Befehls enthalten sind. Für alle Parameter kann eine GUI-Beschreibung (Kap. D.3.2) an der dafür vorgesehenen Stelle in der Parameter-Protokoll-Sequenz definiert werden.
Manche Parametertypen dürfen nur spezielle Identifikationsnummern (IDs) enthalten. Diese werden im Kap. D.3.2 festgelegt.
Das gesamte Protokoll ist zeilenbasiert. Das bedeutet, daß pro Zeile nur eine enweisung eingegeben werden darf und das Ende der Zeile wird auch als das Ende der Anweisung interpretiert.

D.3.2 Syntax des Protokolls

Es folgt die syntaktische Spezifikation des Netzwerkprotokolls. Intern wurde im Hinblick auf internationale Benutzer des DEVISOR-Paketes die Spezifikation des Protokolls auf Englisch erstellt. Hier wird sie in unübersetzter Form wiedergegeben.

Basic data types

INT Integer

GOODSTRING \n-terminated string containing a..z,A..Z,0-9,_,-

EVILSTRING \n-terminated string containing any character

CONSTSTRING \n-terminated string containing A..Z,0-9,_,-

LOCALESTRING \n-terminated string of form xx_YY where xx are two lowercase letters representing a language code in ISO-639 and YY are two uppercase letters representing a country code in ISO-3166

TYPE Integer|Long|Double|Boolean|String

Syntax conventions in this document

[[NAME] Macrosubstitution

{Type:Name} expression 'name' of type 'TYPE' appears once

{Type:Name}* expression 'name' of type 'TYPE' appears at least once

{Type:Name}0 expression 'name' of type 'TYPE' doesn't appear or appears at least once

| binary OR

&& binary AND

CONSTSTRING:TYPE=:MYTYPE new dynamic type definition, when evaluating any expression of type 'MYTYPE', only the type specified by the contant string 'TYPE' is accepted. MYTYPE always is of one of the types defined under "basic data types" above.

Cond?Expr if condition 'cond' holds, insert expression 'Expr'

SIZEOF(Name) number of subblocks of same type in given block 'name'

Protocol structure

```
[HEADER]:=

Header
  Version: {CONSTSTRING:DeViSoR} {INT:majornum}.{INT minornum}
  UserName: {GOODSTRING:id} | {CONSTSTRING:NONE}
  ProjectID: {GOODSTRING:id} | {CONSTSTRING:NONE}
  Subtype: {CONSTSTRING:CONTROL-SERVER|CONTROL-GID|
                        CONTROL-METIS|
                        CONTROL-VISION|VISION-SERVER|
                        CONTROL-GRID
                        CONTROL-NUMERICS}
  MessageType: {CONSTSTRING:REGULAR|ERROR}
  Locale: {LOCALESTRING: locale}
End_Header
```

Codebeispiel 97: Header protocol structure

```
[FOOTER] :=

Footer
  {EVILSTRING:infostring}*
End_Footer
```

Codebeispiel 98: Footer protocol structure

```
[LOE] :=

ListOfErrors
  ErrorHeader
    Type: {CONSTSTRING:CRITICAL|NONCRITICAL}
    Description: {EVILSTRING:infostring}
  End_ErrorHeader

  Error*
    ID: {INT:id}
    Source: {GOODSTRING:modulename}
    InstanceHandle: {GOODSTRING:handle}

    Message
      {EVILSTRING:infostring}*
    End_Message
  End_Error

  ErrorResult
    {EVILSTRING:infostring}*
  End_ErrorResult
End_ListOfErrors
```

Codebeispiel 99: ListOfErrors protocol structure

```
[SEQCONTROL] := [SEQ1] ==> [SEQSERVER] := [SEQ2] | [SEQ3]

[SEQ1] := [HEADER| [CMD] [FOOTER]
[SEQ2] := [HEADER| [LOE] [FOOTER]
[SEQ3] := [HEADER| [CMDREPLY] [FOOTER]
```

Codebeispiel 100: SeqControl protocol structure

Command definitions

```
[CMD]0 :=  (Subtype: CONTROL-SERVER)

Command
  GetRunningModules
End_Command
```

Codebeispiel 101: GetRunningModules protocol command

```
[CMDREPLY]0 :=  (Subtype: CONTROL-SERVER)

GetRunningModulesReply
  Description
    {EVILSTRING:info}*
  End_Description
  Module0
    ID:  {INT:id}
    ProjectID: {GOODSTRING:projectid}
    InstanceHandle: {GOODSTRING:handle}
    Name: {GOODSTRING:name}
    Type: {CONSTSTRING:GRID|METIS|NUMERICS|VISION|GID}
    Description
      {EVILSTRING:info}*
    End_Description
  End_Module
End_GetRunningModulesReply
```

Codebeispiel 102: GetRunningModulesReply protocol command

```
[CMD]0B :=  (Subtype: CONTROL-SERVER)

Command
  GetInactiveModules
End_Command
```

Codebeispiel 103: GetInactiveModules protocol command

```
[CMDREPLY]0B :=  (Subtype: CONTROL-SERVER)

GetInactiveModulesReply
  Description
    {EVILSTRING:info}*
  End_Description
  Module0
    ID:  {INT:id}
    ProjectID: {GOODSTRING:projectid}
    InstanceHandle: {GOODSTRING:handle}
    Name: {GOODSTRING:name}
    Type: {CONSTSTRING:GRID|METIS|NUMERICS|VISION|GID}
    Description
      {EVILSTRING:info}*
    End_Description
  End_Module
End_GetInactiveModulesReply
```

Codebeispiel 104: GetInactiveModulesReply protocol command

```
[CMD]1 :=  (Subtype: CONTROL-SERVER)

Command
  GetAvailableModules
End_Command
```

Codebeispiel 105: GetAvailableModules protocol command

```
[CMDREPLY]1 :=  (Subtype: CONTROL-SERVER)

GetAvailableModulesReply
  Description
    {EVILSTRING:info}*
  End_Description
  Module*
    ID:  {INT:id}
    InstanceHandle: {CONSTSTRING:NONE}
    Name: {GOODSTRING:name}
    Type: {CONSTSTRING:GRID|METIS|NUMERICS|VISION|GID}
    Description
      {EVILSTRING:info}*
    End_Description
  End_Module
End_GetAvailableModulesReply
```

Codebeispiel 106: GetAvailableModulesReply protocol command

```
[CMD]2 :=   (Subtype: CONTROL-SERVER)

Command
  StartModule
  ModuleID: {INT:module-id}
  Display: {EVILSTRING:display}
End_Command
```

Codebeispiel 107: StartModule protocol command

```
[CMDREPLY]2 :=  (Subtype: CONTROL-SERVER)

StartModuleReply
  InstanceHandle: {GOODSTRING:handle}
End_StartModuleReply
```

Codebeispiel 108: StartModuleReply protocol command

```
[CMD]2B :=   (Subtype: CONTROL-SERVER)

Command
  ReStartModule
  ModuleID: {INT:module-id}
  InstanceHandle: {INT:instancehandle}
  Display: {EVILSTRING:display}
End_Command
```

Codebeispiel 109: ReStartModule protocol command

```
[CMDREPLY]2B :=  (Subtype: CONTROL-SERVER)

ReStartModuleReply
  [LOE]
End_ReStartModuleReply
```

Codebeispiel 110: ReStartModuleReply protocol command

```
[CMD]3 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  GetProblemList
  ModuleID: {INT:module-id}
End_Command
```

Codebeispiel 111: GetProblemList protocol command

```
[CMDREPLY]3 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

GetProblemListReply
  Problem*
    ID: {INT:problem-id}
    Name: {GOODSTRING:name}
    Description
      {EVILSTRING:info}*
    End_Description
    LocalProtocol: {CONSTSTRING:NONE|STARTSTOP|
    STARTSTOPPAUSE|STARTSTOPPAUSERECONFIG|FULL}
  End_Problem
End_GetProblemListReply
```

Codebeispiel 112: GetProblemListReply protocol command

```
[CMD]4  := (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  GetParameterListFor
    ModuleID: {INT:module-id}
    ProblemID: {INT:problem-id}
 End_Command
```

Codebeispiel 113: GetParameterListFor protocol command

```
[CMDREPLY]4 := (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

GetParameterListForReply
  {[SCALAR]|[VECTOR][NODE][SELECTION]|
  [GRID]|[FUNCTION]|[DATASOURCE]}*
End_GetParameterListForReply
```

Codebeispiel 114: GetParameterListForReply protocol command

```
[CMD]5 := (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  GetConfigurationListFor
    ModuleID: {INT:module-id}
End_Command
```

Codebeispiel 115: GetConfigurationListFor protocol command

```
[CMDREPLY]5 := (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

GetConfigurationListForReply
  {[SCALAR]|[VECTOR][NODE][SELECTION]|
  [GRID]|[FUNCTION]|[DATASOURCE]}*
End_GetConfigurationListForReply
```

Codebeispiel 116: GetConfigurationListForReply protocol command

```
[CMD]6 :=   (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  GetCompleteConfiguration
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 117: GetCompleteConfiguration protocol command

```
[CMDREPLY]6 :=   (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

GetCompleteConfigurationReply

   Problem
     ID: {INT:problem-id}
     Name: {GOODSTRING:name}
     Description
      {EVILSTRING:info}*
     End_Description
     LocalProtocol: {CONSTSTRING:NONE|STARTSTOP|
     STARTSTOPPAUSE|STARTSTOPPAUSERECONFIG|FULL}
   End_Problem

   GetParameters
     {[SCALAR]|[VECTOR]|[NODE]|[GRID]
     [FUNCTION]|[SELECTION]|[DATASOURCE]}*
    End_GetParameters

    GetConfiguration
      {[SCALAR]|[VECTOR]|[NODE]|[GRID]
      [FUNCTION]|[SELECTION]|[DATASOURCE]}*
    End_SetConfiguration
End_GetCompleteConfigurationReply
```

Codebeispiel 118: GetCompleteConfigurationReply protocol command

```
[CMD]7 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  SetCompleteConfiguration
    ModuleID: {INT:module-id}
    ProblemID: {INT:problem-id}
    InstanceHandle: {INT:handle}

    SetParameters
     {[SETSCALAR]|[SETVECTOR]|[SETNODE]|[SETGRID]
     [SETFUNCTION]|[SETSELECTION]|[SETDATASOURCE]}*
    End_SetParameters

    SetConfiguration
             {[SETSCALAR]|[SETVECTOR]|[SETNODE]|[SETGRID]
             [SETFUNCTION]|[SETSELECTION]|[SETDATASOURCE]}*
    End_SetConfiguration

End_Command
```

Codebeispiel 119: SetCompleteConfiguration protocol command

```
[CMDREPLY]7 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

SetCompleteConfigurationReply
  [LOE]
End_SetCompleteConfigurationReply
```

Codebeispiel 120: SetCompleteConfigurationReply protocol command

```
[CMD]8 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  Start
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 121: Start protocol command

```
[CMDREPLY]8 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

StartReply
 [LOE]
End_StartReply
```

Codebeispiel 122: StartReply protocol command

```
[CMD]9 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  Stop
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 123: Stop protocol command

```
[CMDREPLY]9 := (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

StopReply
 [LOE]
End_StopReply
```

Codebeispiel 124: StopReply protocol command

```
[CMD]10 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  Pause
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 125: Pause protocol command

```
[CMDREPLY]10 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

PauseReply
 [LOE]
End_PauseReply
```

Codebeispiel 126: PauseReply protocol command

```
[CMD]10B :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  Step
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 127: Step protocol command

```
[CMDREPLY]10B :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

StepReply
 [LOE]
End_StepReply
```

Codebeispiel 128: StepReply protocol command

```
[CMD]11 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  Rewind
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
    Steps: {INT:steps}
End_Command
```

Codebeispiel 129: Rewind protocol command

```
[CMDREPLY]11 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

RewindReply
  [LOE]
End_RewindReply
```

Codebeispiel 130: RewindReply protocol command

```
[CMD]12 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  FastForward
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
    Steps: {INT:steps}
End_Command
```

Codebeispiel 131: FastForward protocol command

```
[CMDREPLY]12 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

FastForwardReply
  [LOE]
End_FastForwardReply
```

Codebeispiel 132: FastForwardReply protocol command

```
[CMD]13 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  GetCurrentStatistics
   ModuleID: {INT:module-id}
   InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 133: GetCurrentStatistics protocol command

```
[CMDREPLY]13 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

GetCurrentStatisticsReply
  Statistics0
    Date: {GOODSTRING:date}
    Time: {GOODSTRING:time}
    TimeStep: {INT:timestep}
    {[SCALAR]|[VECTOR]}*
  End_Statistics
End_GetCurrentStatisticsReply
```

Codebeispiel 134: GetCurrentStatisticsReply protocol command

```
[CMD]14 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  GetCompleteStatistics
   ModuleID: {INT:module-id}
   InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 135: GetCompleteStatistics protocol command

```
[CMDREPLY]14 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

GetCompleteStatisticsReply
  Statistics0
    Date: {GOODSTRING:date}
    Time: {GOODSTRING:time}
    TimeStep: {INT:timestep}
    {[SCALAR]|[VECTOR]}*
  End_Statistics
End_GetCompleteStatisticsReply
```

Codebeispiel 136: GetCompleteStatisticsReply protocol command

```
[CMD]15 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
   GetAvailableResults
   ModuleID: {INT:module-id}
   InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 137: GetAvailableResults protocol command

```
[CMDREPLY]15 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

GetAvailableResultsReply
  Result0
    ResultID: {INT:id}
    Name: {GOODSTRING:name}
    Type: {CONSTSTRING:GMV|FEAST|PS|MOVIE|IMAGE|ASCIIDATA}
    Description
      {EVILSTRING:info}*
    End_Description
    BaseFilename: {EVILSTRING: basefilename}
    AvailableDataFields
      {GOODSTRING:gmvlabel | NONE}*
    End_AvailableDataFields
    NumberOfTimeSteps: {INT:nofts}
    AvailableTimeSteps
      {INT:timestep, GOODSTRING:creationtime,
      GOODSTRING:creationdate, LONG:size}0
    End_AvailableTimeSteps
  End_Result
End_GetAvailableResultsReply
```

Codebeispiel 138: GetAvailableResultsReply protocol command

```
[CMD]16 :=   (Subtype: CONTROL-SERVER|VISION-SERVER)

Command
  GetResult
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
    ResultID: {INT:result-id}
    NumberOfTimeSteps: {INT:nots}
    RequestedTimeSteps
      {INT: timestep}0
    End_RequestedTimeSteps
End_Command
```

Codebeispiel 139: GetResult protocol command

```
[CMDREPLY]16 :=   (Subtype: CONTROL-SERVER|VISION-SERVER)

GetResultReply
   [RESULT]0

End_GetResultReply

[RESULT] :=

byte 1: status (0<==>no error)
byte 2--9: size, LSB...MSB
byte 10..10+size: data
```

Codebeispiel 140: GetResultReply protocol command

```
[CMD]17 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  AskForNotify
End_Command
```

Codebeispiel 141: AskForNotify protocol command

```
[CMDREPLY]17 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

AskForNotifyReply
  [STATISTICS] | [LOE] | {CONSTSTRING:IAMALIVE}
End_AskForNotifyReply
```

Codebeispiel 142: AskForNotifyReply protocol command

```
[CMD]18:=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  CloseModule
   ModuleID: {INT:module-id}
   InstanceHandle: {INT:handle}
EndCommand
```

Codebeispiel 143: CloseModule protocol command

```
[CMDREPLY]18 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

CloseModuleReply
  [LOE]
End_CloseModuleReply
```

Codebeispiel 144: CloseModuleReply protocol command

```
[CMD]19:= (Subtype: CONTROL-SERVER)

Command
  DisconnectModule
   ModuleID: {INT:module-id}
   InstanceHandle: {INT:handle}
EndCommand
```

Codebeispiel 145: DisconnectModule protocol command

```
[CMDREPLY]19 :=   (Subtype: CONTROL-SERVER)

DisconnectModuleReply
  [LOE]
End_DisconnectModuleReply
```

Codebeispiel 146: DisconnectModuleReply protocol command

```
[CMD]20:=   (Subtype: CONTROL-SERVER)

Command
  ConnectModule
   ModuleID: {INT:module-id}
   InstanceHandle: {INT:handle}
EndCommand
```

Codebeispiel 147: ConnectModule protocol command

```
[CMDREPLY]20 :=   (Subtype: CONTROL-SERVER)

ConnectModuleReply
  [LOE]
End_ConnectModuleReply
```

Codebeispiel 148: ConnectModuleReply protocol command

```
[CMD]21 :=   (Subtype: CONTROL-SERVER)

Command
  ClearResult
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
    ResultID: {INT:result-id}
    NumberOfTimeSteps: {INT:nots}
    SelectedTimeSteps
      {INT: timestep}0
    End_SelectedTimeSteps
End_Command
```

Codebeispiel 149: ClearResult protocol command

```
[CMDREPLY]21 :=   (Subtype: CONTROL-SERVER)

ClearResultReply
  [LOE]
End_ClearResultReply
```

Codebeispiel 150: ClearResultReply protocol command

```
[CMD]22 :=   (Subtype: CONTROL-SERVER)

Command
  ClearProject
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 151: ClearProject protocol command

```
[CMDREPLY]22 :=    (Subtype: CONTROL-SERVER)

ClearProjectReply
  [LOE]
End_ClearProjectReply
```

Codebeispiel 152: ClearProjectReply protocol command

```
[CMD]23 :=   (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  UpdateRouterTable
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 153: UpdateRouterTable protocol command

```
[CMDREPLY]23 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

UpdateRouterTableReply
  [LOE]
End_UpdateRouterTableReply
```

Codebeispiel 154: UpdateRouterTableReply protocol command

```
[CMD]24 :=

Command
  ExitServer
End_Command
```

Codebeispiel 155: ExitServer protocol command

```
[CMDREPLY]24 :=

ExitServerReply
End_ExitServerReply
```

Codebeispiel 156: ExitServerReply protocol command

```
[CMD]25 :=   (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|
                     CONTROL-GID|CONTROL-METIS)

Command
  AskForLocalPort
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 157: AskForLocalPort protocol command

```
[CMDREPLY]25 :=  (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|CONTROL-GID|
                     CONTROL-METIS)

AskForLocalPortReply
  Port: {INT:port}
End_AskForLocalPortReplyReply
```

Codebeispiel 158: AskForLocalPortReply protocol command

```
[CMD]26 := (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|CONTROL-GID|
                     CONTROL-METIS)

Command
  GetStatus
    ModuleID: {INT:module-id}
    InstanceHandle: {INT:handle}
End_Command
```

Codebeispiel 159: GetStatus protocol command

```
[CMDREPLY]26 := (Subtype: CONTROL-NUMERICS|CONTROL-GRID|
                     CONTROL-VISION|CONTROL-GID|
                     CONTROL-METIS)

GetStatusReply
  Status: {CONSTSTRING: UNKNOWN | STARTED | CONFIGURED |
                        RUNNING | FINISHED}
  TapeRecorder: {CONSTSTRING: PAUSE | PLAY | STOP}
  CurrentTimeStep: {INT:timestep}
  ResultsAvailable: {Boolean:available}
End_GetStatusReply
```

Codebeispiel 160: GetStatusReply protocol command

Parameterdefinition

```
[NODE]:=

NodeParameter
  ID: {INT:parameter-id}
  Name: {GOODSTRING:parameter-name}
  ShortName: {GOODSTRING: shortname}
  Reconfigurable: {CONSTSTRING:TRUE|FALSE}
  Description
    {GOODSTRING:text}*
  End_Description
  {[SCALAR]|[VECTOR]|[GRID]|[FUNCTION]|[SELECTION]|[NODE]}*
End_NodeParameter
```

Codebeispiel 161: NodeParameter protocol definition

```
[SETNODE]:=

SetNodeParameter
  ID: {INT:parameter-id}
  {[SETSCALAR]|[SETVECTOR]|[SETGRID]|
  [SETFUNCTION]|[SETSELECTION]|[SETNODE]}*
End_SetNodeParameter
```

Codebeispiel 162: SetNodeParameter protocol definition

```
[SCALAR]:=

ScalarParameter
  ID: {INT:parameter-id}
  Name: {GOODSTRING:parameter-name}
  ShortName: {GOODSTRING: shortname}
  Reconfigurable: {CONSTSTRING:TRUE|FALSE}
  Description
    {GOODSTRING:text}*
  End_Description
  Format: {CONSTSTRING:TYPE}
  MinValue: {TYPE:minimum}
  MaxValue: {TYPE:maximum}
  Value: {TYPE:default} | {CONSTSTRING:NONE}
  Unit: {GOODSTRING:unitname}
  {
     {{Format=Integer} ? [GUITEXTFIELD]|[GUISLIDER]|
                         [GUIRADIOBUTTON]|[GUICHECKBOX]} |
     {{Format=Double}  ? [GUITEXTFIELD]|[GUISLIDER]|
                         [GUIRADIOBUTTON]|[GUICHECKBOX]}|
     {{Format=String}  ? [GUITEXTFIELD]|[GUITEXTAREA]|
                         [GUICHECKBOX]|[GUIRADIOBUTTON]} |
     {{Format=Boolean} ? [GUICHECKBOX]}
  }
End_ScalarParameter
```

Codebeispiel 163: ScalarParameter protocol definition

```
[SETSCALAR]:=

SetScalarParameter
  ID: {INT:parameter-id}
  Value: {TYPE:default}
End_SetScalarParameter
```

Codebeispiel 164: SetScalarParameter protocol definition

```
[VECTOR]:=

VectorParameter
  ID: {INT:parameter-id}
  Name: {GOODSTRING:parameter-name}
  ShortName: {GOODSTRING: shotname}
  Reconfigurable: {CONSTSTRING:TRUE|FALSE}
  Description
    {GOODSTRING:text}*
  End_Description
  Format: {CONSTSTRING:TYPE =: MYTYPE}
  Values
    Entry*
      Label: {GOODSTRING:label}
      Value: {MYTYPE:value}
    End_Entry
  End_Values
  Unit: {GOODSTRING:unitname}
  {
     {{Format=Integer} ? [GUITEXTFIELD]|[GUISLIDER]|
                         [GUIRADIOBUTTON]|[GUICHECKBOX]} |
     {{Format=Double}  ? [GUITEXTFIELD]|[GUISLIDER]|
                         [GUIRADIOBUTTON]|[GUICHECKBOX]}|
     {{Format=String}  ? [GUITEXTFIELD]|[GUITEXTAREA]|
                         [GUICHECKBOX]|[GUIRADIOBUTTON]} |
     {{Format=Boolean} ? [GUICHECKBOX]}
     {{Format=Integer && SIZEOF(Values)=3} ?
                         [GUICOLORCHOOSER]}
  }
End_VectorParameter
```

Codebeispiel 165: VectorParameter protocol definition

```
[SETVECTOR]:=

SetVectorParameter
  ID: {INT:parameter-id}
  Values
    Entry*
      Value: {GOODSTRING:value}
    End_Entry
  End_Values
End_SetVectorParameter
```

Codebeispiel 166: SetVectorParameter protocol definition

```
[GRID]:=

GridParameter
  ID: {INT:parameter-id}
  Name: {GOODSTRING:parameter-name}
  ShortName: {GOODSTRING: shortname}
  Reconfigurable: {CONSTSTRING:TRUE|FALSE}
  Description
    {GOODSTRING:text}*
  End_Description
  Size: {LONG:size}
  Value
    [FEAST]
  End_Value
  [GUIGRIDVIEWER]
End_GridParameter
```

Codebeispiel 167: GridParameter protocol definition

```
[SETGRID]:=

SetGridParameter
  ID: {INT:parameter-id}
  Size: {LONG:size}
  Value
    [FEAST]
  End_Value
End_SetGridParameter
```

Codebeispiel 168: SetGridParameter protocol definition

```
[FUNCTION]:=

FunctionParameter
  ID: {INT:parameter-id}
  Name: {GOODSTRING:parameter-name}
  ShortName: {GOODSTRING: shortname}
  Reconfigurable: {CONSTSTRING:TRUE|FALSE}
  Description
    {GOODSTRING:text}*
  End_Description
  GlobalParameters
    {[TYPE]: {GOODSTRING:varname}}0
  End_GlobalParameters
  Function
    {[TYPE]: {GOODSTRING:varname}}0
    {[ASSIGNMENT] | [IFBLOCK]}*
  End_Function
  [GUIFUNCTIONEDITOR]
End_FunctionParameter
```

Codebeispiel 169: FunctionParameter protocol definition

```
[SETFUNCTION]:=

SetFunctionParameter
  ID: {INT:parameter-id}
  GlobalParameters
    {[TYPE]: {GOODSTRING:varname}}0
  End_GlobalParameters
  Function
     {[TYPE]: {GOODSTRING:varname}}0
    {[ASSIGNMENT] | [IFBLOCK]}*
End_SetFunctionParameter
```

Codebeispiel 170: SetFunctionParameter protocol definition

```
[ASSIGNMENT] :=  [VAR] := [ASSIGNMENT0];

[ASSIGNMENT0] := {[ASSIGNMENT0] | {INT} | {DOUBLE} |
                  [FUNC2]([ASSIGNMENT0],[ASSIGNMENT0])
                  | [FUNC]([ASSIGNMENT0]) | [VAR] | [CONST]}
                  {*|/|-|+}{[ASSIGNMENT0] | {INT}
                          | {DOUBLE} | [FUNC2]([ASSIGNMENT0],
                          [ASSIGNMENT0])
                  | [FUNC]([ASSIGNMENT0]) | [VAR] | [CONST]}

[FUNC] := {LN  | SQRT | EXP}

[FUNC2] := {POW | MIN | MAX}

[CONST] := { PI | E }

[VAR]:= {GOODSTRING:varname}

[EQN] :=  [ASSIGNMENT0]{>,==,<,<=,>=,!=}[ASSIGNMENT0]

[IFBLOCK]:=

IF
 [EQN]
THEN
{[ASSIGNMENT]|[IFBLOCK]}*
ELSE
{[ASSIGNMENT]|[IFBLOCK]}*
ENDIF



IF
 [EQN]
THEN
{[ASSIGNMENT]|[IFBLOCK]}*
ENDIF
```

Codebeispiel 171: Allowed Functions

```
[SELECTION]:=

SelectionParameter
  ID: {INT:parameter-id}
  Name: {GOODSTRING:parameter-name}
  ShortName: {GOODSTRING: shotname}
  Reconfigurable: {CONSTSTRING:TRUE|FALSE}
  Description
    {GOODSTRING:text}*
  End_Description
  MinimumSets: {INT:minimumsets}
  MaximumSets: {INT:maximumsets}
  {[SCALAR]|[VECTOR]|[GRID]|[FUNCTION]|[SELECTION]|[NODE]}*
  SelectedIds
  {INT id}*
  End_SelectedIds
  [GUISELECTION] | [GUIRADIOBUTTON] | [GUICHECKBOX] |
  [GUICOMBOBOX]
End_SelectionParameter
```

Codebeispiel 172: SelectionParameter protocol definition

```
[SETSELECTION]:=

SetSelectionParameter
  ID: {INT:parameter-id}
  {[SETSCALAR]|[SETVECTOR]|[SETGRID]|
  [SETFUNCTION]|[SETSELECTION]}*
End_SetSelectionParameter
```

Codebeispiel 173: SetSelectionParameter protocol definition

```
[DATASOURCE]:=

DataSourceParameter
  ID: 2
  Host: {GOODSTRING:host}
  Port: {INT:port}
  ModuleID: {INT:moduleid}
  InstanceHandle: {GOODSTRING:instanceHandle}
  ResultID: {INT:resultid}
  DataField: {GOODSTRING:gmvlabel}
  SelectedTimeSteps
    {INT:timestep}0
  End_SelectedTimeSteps
End_DataSourceParameter

[SETDATASOURCE] := [DATASOURCE]
```

Codebeispiel 174: DataSourceParameter protocol definition

GUI-Definition

```
[GUITEXTFIELD]:=

GUI
  TextField
End_GUI
```

Codebeispiel 175: GUI TextField protocol definition

```
[GUITEXTAREA]:=

GUI
  TextArea
End_GUI
```

Codebeispiel 176: GUI TextArea protocol definition

```
[GUISLIDER]:=

GUI
  Slider
    From: {INT:from}|{DOUBLE:from}
    To:   {INT:to}|{DOUBLE:to}
    Step: {INT:step}|{DOUBLE:step}
    Mode: {CONSTSTRING:CONTINUOUS|DISCRETE}
    MajorTicksSpacing: {INT major}
    MinorTicksSpacing: {INT minor}
End_GUI
```

Codebeispiel 177: GUI Slider protocol definition

```
[GUIRADIOBUTTON]:=

GUI
  RadioButton
    Default: {INT:default}
    Button*
      ID: {INT:default}
      Name: {GOODSTRING:name}
      Value: {INT:value}|{DOUBLE:value}|{STRING:value}
    End_Button
End_GUI
```

Codebeispiel 178: GUI RadioButton protocol definition

```
[GUICHECKBOX]:=

GUI
  CheckBox
End_GUI
```

Codebeispiel 179: GUI CheckBox protocol definition

```
[GUICOMBOBOX]:=

GUI
  ComboBox
    Editable: {Boolean: editable}
End_GUI
```

Codebeispiel 180: GUI ComboBox protocol definition

```
[GUICOLORCHOOSER]:=

GUI
  ColorCooser
End_GUI
```

Codebeispiel 181: GUI ColorChooser protocol definition

```
[GUISELECTION]:=

GUI
  Selection
End_GUI
```

Codebeispiel 182: GUI Selection protocol definition

```
[GUIGRIDVIEWER]:=
GUI
  GridViewer
End_GUI
```

Codebeispiel 183: GUI GridViewer protocol definition

```
[GUIFUNCTIONEDITOR]:=
GUI
  FunctionEditor
End_GUI
```

Codebeispiel 184: GUI FunctionEditor protocol definition

Special parameter IDs

Id	**Meaning**	**Structure**
1	StatisticsAvailable	SelectionParameter with list of ScalarParameters only
2	Data source for modules	[DATASOURCE] only
< 1024	per Definition reserved IDs	

Tabelle D.1: Special parameter Id’s

Anhang E

GNU General Public Licence

Version 2, June 1991
Copyright © 1989, 1991 Free Software Foundation, Inc.

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.
Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.
The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice

and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.
To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

> one line to give the program’s name and a brief idea of what it does.
> Copyright (C) yyyy name of author
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.
If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) yyyy name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
> This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.
You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.
>
> signature of Ty Coon, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Literaturverzeichnis

[1] ABRAMOWSKI, S. und MÜLLER, H.: *Geometrisches Modellieren*. BI-Wissenschaftsverlag, Jan. 1992.

[2] AVS: *Advanced Visualization Systems*. Techn. Ber., Advanced Visualization Systems, Jan. 2003. http://www.avs.com.

[3] BECKER, CH. und TUREK, S.: *FEATFLOW Finite Element Software for the Incompressible Navier-Stokes equations - User Manual Release 1.1*. Techn. Ber., Universität Heidelberg, Jan. 1998.

[4] BERN, M. und EPPSTEIN, D.: *Mesh Generation and Optimal Triangulation*. UC Irvine Dept. Inf. & Computer Science, Jan. 1995. http://www.ics.uci.edu/~eppstein/pubs/BerEpp-CEG-95.pdf.

[5] BORLAND: *TogetherJ 6.0*. Techn. Ber., TogetherJ 6.0, Jan. 2002. www.togethersoft.com.

[6] BRODLIE ET AL.: *Scientific Visualization*. Springer Verlag, Jan. 1992.

[7] CEDERQUIST: *Version Management with CVS*. Cederqvist et al, Jan. 2003. http://www.cvshome.org/docs/manual/cvs.html.

[8] COMER, D. E.: *Computer Networks and Internets*. Prentice-Hall, Jan. 1999.

[9] COMSOL: *FEMLAB 2.3*. Comsol Group, Jan. 2003. http://www.femlab.com.

[10] DARMOFAL und HAIMES: *An Analysis of 3-D Particle Path Integration Algorithms*. In: *AIAA Paper* . San Diego CA, Jan. 1995. htttp://raphael.mit.edu/pv3/pv3.html.

[11] DEUFLHARD und BORNEMANN: *Numerische Mathematik II - Integration gewöhnlicher Differentialgleichungen*. de Gruyter, Jan. 1994.

[12] FLUENT: *Fluent 6.1*. Fluent Inc., Jan. 2002. http://www.fluent.com.

[13] FOLEY, J. D., VAN DAM, A., FEINER, S. und HUGHES: *Computer Graphics- Principles and Practice (2nd Edition)*. Addison-Wesley, Jan. 1996.

[14] GNU: *Free Software Foundation: The GNU General Public Licence (GPL)*. Techn. Ber., Free Software Foundation: The GNU General Public Licence (GPL), Jan. 1991. http://www.gnu.org/licenses/Licenses.html#GPL .

[15] GONZALEZ, R. C. und WOODS, R. E.: *Digital Image Processing*. Addison-Wesley, Jan. 1992.

[16] GROSSMANN, CH. und ROOS, H.-G.: *Numerik partieller Differentialgleichungen*. Teubner, 1994. 2.Auflage.

[17] HAIRER, NORSETT und WANNER: *Solving Ordinary Differential Equations I*, Bd. I - Nonstiff Problems. Springer Verlag, Jan. 1987.

[18] HAIRER, NORSETT und WANNER: *Solving Ordinary Differential Equations II*, Bd. II - Stiff Problems. Springer Verlag, Jan. 1992.

[19] HIRSCH, C.: *Numerical computation of internal and external flows, Band 1*. Wiley, 2000.

[20] JABLONOWSKI, D., BRUNET, J., BLISS, B. und HABER, R.: *VASE: the Visualization and Application Steering Environment*. In: *SuperComputing 1993*, Jan. 1993.

[21] KÖSTER, M.: *FEATFLOW in a Windows environment*. Techn. Ber., Universität Dortmund, Vogelpothsweg 87, 2002.

[22] KRUSKA: *Wissenschaftliche Visualisierung, Skript zur Vorlesung*. http://phong.informatik.uni-leipzig.de/kruska/visuallecture.html, Jan. 2000.

[23] LANE: *Scientific Visualization*, Kap. Scientific Visualizaiton of Large Scale Unsteady Fluid Flows. IEEE Computer Society, Jan. 1997.

[24] LUBER und HASTREITER: *Script of Lecture Scientific Visualization*. Universität Erlagen, Jan. 1999.

[25] NUMERICAL ALGORITHMS GROUP: *IRIS Explorer*. Techn. Ber., IRIS Explorer, Jan. 2003. www.nag.co.uk.

[26] ORTEGA, F. A.: *The General Mesh Viewer Version 3.2*. Los Alamos National Laboratory, Jan. 2003. http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html.

[27] PINNAU, R.: *Höhere Numerik II - Theorie und Numerik elliptischer Differentialgleichungen*. Vorlesungsskript, TU Darmstadt, 2001.

[28] QUICKTIME: *Apple Computer Inc.: QuickTime Documentation*. Techn. Ber., Apple Computer Inc.: QuickTime Documentation, Jan. 2003. http://developer.apple.com/documentation/QuickTime/QuickTime.html.

[29] RANNACHER, R.: *Numerik gewöhnlicher Differentialgleichungen*. http://gaia.iwr.uni-heidelberg.de/httpdoc/Lectures/Notes/notes.html, Jan. 2001.

[30] RED HAT: *Cygwin 1.3*, Jan. 2003. http://www.cygwin.com.

[31] SARDAJOEN, VAN WALSUM, HIN und POST: *Scientific Visualization*, Kap. Particle Tracing Algorithms for 3D Curvilinear Grids. IEEE Computer Society, Jan. 1997.

[32] SCHWARZ, H. R.: *Methode der finiten Elemente,2. Auflage*. Stuttgart: Teubner, 1984.

[33] SCHWARZ, H.R.: *Numerische Mathematik*. Teubner, 1997.

[34] SILBERSCHATZ, A., GAGNE, G. und GALVIN, P.B.: *Operating Systems Concepts (6th Edition)*. John Wiley & Sons, Jan. 2002.

[35] SWING: *Java allgemein*. Techn. Ber., Sun Microsysteme, Jan. 1996. `http://java.sun.com/docs/books/tutorial/uiswing/index.html`.

[36] TANENBAUM, A.: *Computer Networks (4th Edition)*. Prentice Hall, Jan. 2002.

[37] TUREK, S.: *FEATFLOW . Finite element software for the incompressible Navier-Stokes equations: User Manual, Release 1.1*. Technical report, 1998.

[38] TUREK, S.: *Numerische Methoden für Partielle Differentialgleichungen*. Vorlesungsskript, Fachbereich Mathematik, Universität Dortmund, 2001.

[39] TUREK, S.: *Numerik 2, Vorlesung an der Universität Dortmund*. WS 2001/2002, basierend auf Rannacher, Jan. 2002.

[40] TUREK, S. und BECKER, CH.: *FEATFLOW - Finite element software for the incompressible Navier–Stokes equations*. User Manual, Universität Dortmund, 1999.

[41] VARIOUS: *Java allgemein*. Techn. Ber., Sun Microsystems, `http://java.sun.com/docs/books/tutorial`, 1996.

[42] VARIOUS: *Java3D*. Techn. Ber., Sun Microsystems, `http://java.sun.com/products/java-media/3D/index.html`, 1996.